UTILITY-DRIVEN OPTIMIZATION AND PLACEMENT FRAMEWORK FOR VISUAL IOT ANALYTICS OVER EDGE-CLOUD ENVIRONMENTS

BY

TAREK ELGAMAL

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Klara Nahrstedt, Chair
Professor Indranil Gupta
Assistant Professor Tianyin Xu
Dr. Rittwik Jana, AT&T Labs

ABSTRACT

Internet of Things (IoT) applications generate massive amounts of real-time data. A large amount of this data is visual data that comes from cameras. Reports by Information Handling Services (IHS) indicate that 245 million professionally installed surveillance cameras are operating worldwide as of 2015 [1]. We refer to the data coming from such cameras as Visual IoT data. Recent advances in computer vision and neural networks have made it possible for more visual IoT data to be automatically searched and analyzed by algorithms rather than humans. This happens in parallel with advances in Edge computing and Serverless computing. *Edge computing* [2], has emerged to allow analyzing visual IoT data closer to where it is generated, and hence avoiding sending vast amounts of visual data streams to be analyzed in one remote location. On the other hand, serverless computing facilitates the analysis of such streams by allowing users to deploy individual analysis functions in user-owned edge devices or public cloud infrastructure.

In this dissertation, we argue that the current video analytics systems are not keeping up with such advances. For example, video encoders have been designed for a long time to please human viewers and be agnostic of the downstream analysis tasks (e.g., object detection). Moreover, existing video analytics systems fail to leverage pipeline parallelism when distributing the analysis across edge and cloud devices. Existing systems also do not address several challenges associated with deploying analytics functions on public cloud infrastructure. Such challenges include performing hybrid edge and cloud analytics in a price-efficient manner as well as protecting the privacy and confidentiality of users' sensitive data against misuse by the edge/cloud provider.

We address the above challenges by: (1) building a framework for processing visual data streams across edge and cloud compute resources, (2) developing algorithms that identify the best placement of computations across edge and cloud resources to optimize various utilities (e.g., latency, bandwidth, price, and privacy), and (3) building the systems that validate the effectiveness of the optimization algorithms and their ability to control the tradeoff between different utilities. The framework and the algorithms optimize various utilities and address the tradeoffs between them. The first algorithm focuses on optimizing the **bandwidth** by detecting the events of interest in videos closer to where the video is generated. To achieve this, we develop a *Semantic Video Encoding* technique in which we redesign the video compression algorithms at the camera to be aware of the edge-based downstream analysis tasks. This allows *compressed videos* to be easily analyzed by algorithms rather than humans because the downstream tasks can search the compressed video for the parts that are relevant to the overall analysis goals.

The second algorithm focuses on optimizing the application's **end-to-end latency**. To achieve

this, we develop an *Operator Placement* algorithm that is given a processing job expressed in the form of a Directed Acyclic Graph (DAG) of operators/functions, it finds which operators to place on an edge device and which operators to place on a remote cloud server. The third algorithm is a **price** optimization algorithm which *optimizes the price of deploying visual IoT analytics applications in serverless computing platforms (e.g. AWS Lambda)*.

The fourth algorithm focuses on optimizing the end-to-end latency of computation while maintaining data **privacy**. To achieve this, we leverage trusted execution environments (e.g., Intel-SGX) which allow users to execute machine learning predictions on visual IoT data while maintaining data confidentiality. To speed up the machine learning predictions, we develop a technique to find the best *partitioning of neural networks* computation across multiple trusted execution environments.

ACKNOWLEDGMENTS

First and foremost, I want to thank Allah for blessing me with the strength and endurance to complete this dissertation, and I want to congratulate my lovely wife Shaima and my awesome son Ziad for this achievement. They have in this degree as much as I do and maybe more. They put up with me during these past years and we have successfully made it together. I got the admission letter to the University of Illinois when me and Shaima were in the hospital giving birth to Ziad and that says alot about the journey that the three of us went through.

Now that I have mentioned my partners in the degree, I would like to express my deepest gratitude to the person who had made this dissertation possible, my advisor Professor Klara Nahrstedt. I am very grateful for all the help, support, and encouragement that she provided throughout the course of this dissertation. I have learned immensely from her both from the technical and personal side. I have always looked up to Klara as an impressive researcher and advisor, and after working closely with her, she also became my role model of how a true leader should look like. Klara has this exceptional skill of giving her students the steering wheel of their own research while making sure that they keep going in a straight line and they do not make any detours or wrong turns. I am incredibly grateful for being a part of her research group and I hope that someday me and her previous graduate students come together and celebrate with her the impact she had made on us.

I would like to sincerely thank the rest of my committee members, Professor Indranil Gupta (Indy), Assistant Professor Tianyin Xu and Dr. Rittwik Jana for their insightful feedback and discussions that had an enormous benefit to my dissertation's work. Indy has been a role model teacher and researcher since my first week in school. You can immediately realize how Indy is inspiring when you walk in his first class. His incredible knowledge as a researcher and his ability to effortlessly explain complex concepts in class is exceptional. I was very lucky to learn from him as a researcher throughout his feedback on my dissertation, and as a teacher when I was a Teaching Assistant of his very popular distributed system class. I was very lucky to have Tianyin in my committee. The energy that Tianyin brought to the department is incredible. You can immediately realize how knowledgeable, humble and encouraging Tianyin is the moment you meet him. I benefited a lot from his feedback on my dissertation and he set a bar for me for how a newly hired Professor can make such a great impact on both the students and the department in such a small time. I would like to sincerely thank Rittwik for the opportunity of interning at AT&T labs. Meeting Rittwik and his group was a turning point in my dissertation. I am grateful for the freedom and empowerment he provided me to expand my dissertation's ideas and apply them to impactful applications that are relevant to AT&T's business. It was a pleasure to have a researcher

of Rittwik's calibre in my Ph.D. committee.

I have also had the chance to work with amazing collaborators during my journey. Matthias Boehm, Berthold Reinwald from IBM research, set a high standard for me on how impactful my research should be. My mentors in AT&T, Shu Shi and Varun Gupta, were a great help and support for making my work relevant to real-production scenarios. I would like to dedicate special thanks to my friend and collaborator Atul Sandur and his advisor Prof. Gul Agha. Atul always knew how to motivate me to work better and harder. He has a great attention to details and has always come up with great questions to challenge my ideas which was extremely helpful and insightful. An important portion of this dissertation is a product of my collaboration with Atul and I want to sincerely thank him.

I would like to express my dearest love and appreciation to my family. I can't thank my wife Shaima and my son Ziad for all the support and sacrifices they have made to make this happen. Shaima has always been there to provide me with tons of love and support in each challenge I faced. Nothing would have been possible without her being by my side. I am so proud to see our son turn into a great young man whose age (5 years) is equivalent to the Ph.D. years I spent.

At the beginning I was disappointed that I will have to do my Ph.D. thesis defense on Zoom but I was so lucky to have my parents (Mohamed El-Gamal and Nadia Emara) and siblings joining in. It was a special moment for our entire family. I am so grateful for their love and support. I am proud to follow in my dad's footsteps (he got his Ph.D. exactly 30 years ago) and my mom is the common factor in both successes. She is the reason for every good thing that happens to the family. Also, I am very lucky to have the most wonderful siblings one can ask for (Amr Elgamal and Heba El-Gamal).

I can never forget the love and support that I got from my parent's in law from the first day I got admitted until the day of the defense. They were in the same room with me when I got the admission letter and their prayers were always with me ever since. I am so blessed to have such supportive in-laws (Abdul Majeed Habeebullah, Gulnaz Azmi, Mustafa Abdul Majeed, and Mohammed Abdul Majeed).

I am thankful to the National Science Foundation and Aerospace Corporation for funding my work. The Aerospace Corporation's University Partnership Program has been a great support for me to finish my Ph.D. dissertation. I sincerely thank Mikhail Tadjikov, John Maguire, and Ingrid Guch who are members of The Aerospace Corporation for their valuable feedback on my work. I am grateful to the Computer Science Department at University of Illinois, Urbana Champaign, for giving me the opportunity to pursue a Ph.D. degree at this great institution. Many thanks to the staff of the department, including but not limited to: Viveka Perera Kudaligama, Kathy Runck, Kara MacGregor, Mary Beth Kelley, Maggie Metzger Chappell, and Samantha Hendon for their help and support during my Ph.D. degree.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 VISUAL IOT APPLICATIONS

Many key applications of the Internet of Things (IoT) process a large amount of visual data streams coming from cameras. Estimates suggest that clusters of cameras on board of a smart vehicle are going to generate 3 TB of data per day and reports by Information Handling Services (IHS) indicate that 245 million professionally installed surveillance cameras are operating worldwide as of 2015 [1]. Analyzing live video streams from those cameras is of considerable importance for decision making in many organizations such as autonomous vehicle companies, traffic departments, police departments, and private security departments. Figure 1.1 shows several examples of visual IoT data sources and their corresponding applications. The data sources have the same properties of IoT devices in which they have sensing and transmission capability without requiring human intervention, however the data these devices produce is in a visual form (i.e., video, image, motion). We focus on visual data because they represent a challenging category of IoT data in terms of: (1) the tremendous bandwidth required to transmit the data, (2) the significant computation required to analyze the data, and (3) the strict latency constraints to preserve the real-time property of the applications. As shown in Figure 1.1, visual IoT data can have a variety of sources (e.g., wearable cameras, scientific instruments,..etc), that are used in many applications (e.g., traffic analysis, stroke detection, motion analysis), and applied in different industries (e.g., healthcare, material science). The examples in Figure 1.1 are some of the applications developed during the course of this dissertation and they inspired the research challenges that are addressed in the dissertation.

1.2 ENABLING TECHNOLOGIES

1.2.1 Machine Learning and Computer Vision

Relying on human labor to visually analyze image/video streams is becoming increasingly infeasible in terms of cost and speed. Hence, automating the analysis of videos is becoming more and more critical especially with the unprecedented amount of visual data streams captured every day across the planet. Studies show that there is one professional camera installed for every 29 people on Earth [3], and it is forecasted that the number of cameras will increase by 20% each year for the next five years. Recent advances in **machine learning and computer vision** [4][5][6] have made it possible to automate the analysis on visual streams with incredible accuracy that often competes

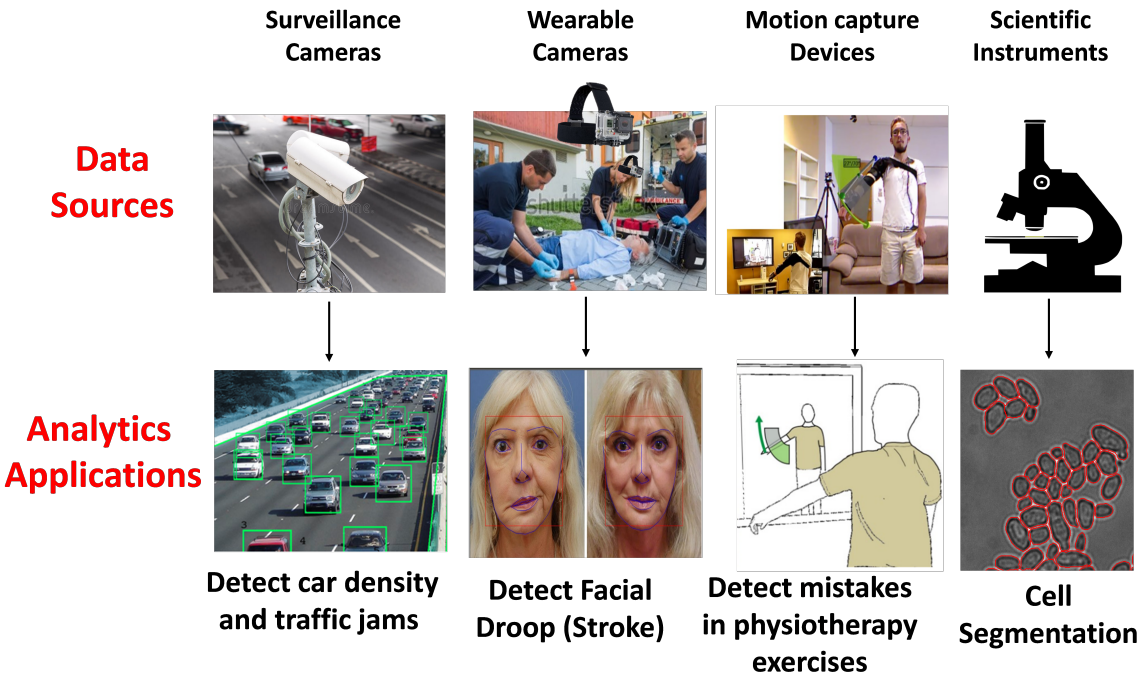

Figure 1.1: Visual IoT data sources and applications.

with human capabilities in several tasks ranging from object classification [7][6] to image-based cancer diagnosis [8][9]. Performing analysis on videos have largely been dependent on machine learning. The literature offers a wealth of proposals for computer vision techniques that are based on machine learning algorithms (e.g., least-squares regression, Support Vector Machines (SVM)) [10][11][12]. Many of these techniques have been displaced by deep neural networks due to the neural network's significant ability to express complex non-linear relationships which results in accuracy that rivals human abilities. However, this expressiveness comes at a cost, which is that neural networks require several orders of magnitudes more labelled data than classical approaches to achieve high classification accuracy. Hence, the choice of the best algorithm ultimately depends on the task and the amount of available data. Regardless of which algorithm one uses, when the data has ground truth labels (e.g., objects), all ML algorithms have a common property which is that the algorithm runs in two phases: a training phase, and an inference/prediction phase. We describe each phase as follows:

Machine learning (ML) training: The training of machine learning models consists of fitting appropriate values to a given set of parameters such that the overall empirical error on a given set of labeled training data is minimized. The number of parameters and their relationship is dependent on the type of the model. For example, a neural network model consists of multiple layers and each layer consists of an arbitrary number of parameters, while the number of parameters of

SVM models is equivalent to the number of features extracted from the image. The process of fitting appropriate values to these parameters is computationally expensive, but is now supported by a wide range of software frameworks such as: TensorFlow, PyTorch, and Scikit learn. To train an object detector (e.g., car) on video, we would first label a portion of a video (or set of videos) by hand, marking which frames contained cars and which did not. Each frame is then fed to a machine learning training framework. After a few iterations and a stopping criteria, the training procedure stops and the latest values of the model parameters are saved. The training phase is a computationally intensive offline phase. Due to the computational cost of training machine learning models, especially neural networks, companies and researchers have also published hundreds of pre-trained models, each representing thousands of hours of CPU and GPU training time.

Machine Learning (ML) Inference: ML inference on videos consists almost exclusively of passing individual video frames or a group of video frames to a pre-trained ML model. That is, to detect objects in video, we evaluate the NN repeatedly, once per frame or per group of frames. The ML inference is less computationally intensive than ML training, however, the challenges in the ML inference come from the real-time nature of video analytics application where inference has to be performed on cameras that capture videos at a high frequency rate (e.g., 30 fps) and high resolution (e.g., 1080p).

In this dissertation, we focus on the analysis operations that are performed near-real time. This description applies to ML inference from pre-trained models rather than ML training. The analysis operations that we support are not limited to ML inference and we support other analysis operations such as feature extraction, clustering, and dimensionality reduction. However, unless explicitly mentioned, the ML workloads in this dissertation belong to the ML inference category. We also note that ML inference can largely benefit from the close proximity of edge devices as we will describe in more detail in the next section.

1.2.2 Edge Computing

Due to the limited computational capability of the camera devices, the conventional approach of performing analysis on visual IoT streams is to send the streams to a centralized data center (cloud) and leverage its powerful and seemingly abundant resources to execute the analytics remotely. However, given the tremendous amount of data transfer required by video data, the latency and bandwidth requirements become significantly high (e.g., 300 GB/month for Nest cameras [13]). For this reason, the concept of cloudlet/edge computing [14] has emerged in which an additional computing layer sits between the camera and the cloud, and is used to help reduce the bandwidth and latency by performing the entire computations on behalf of the cloud or performing simple computations to filter the amount of data being transmitted to the cloud. The additional

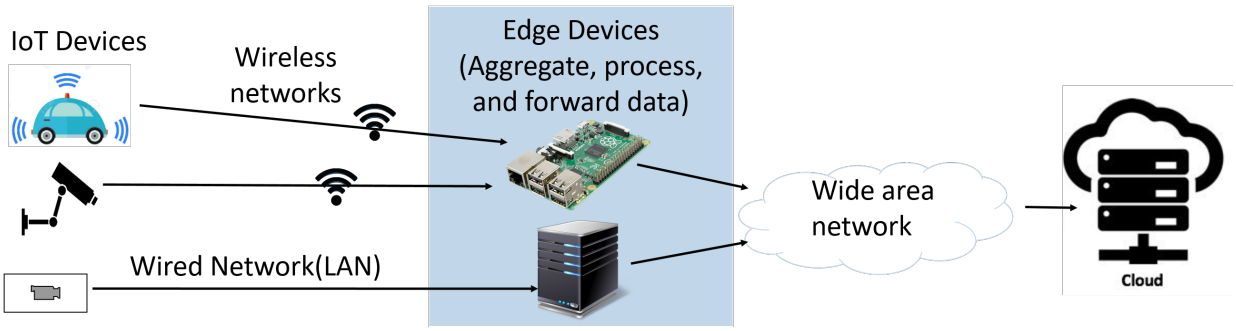

Figure 1.2: IoT edge and cloud infrastructure (3-tier architecture).

compute devices known as edge devices complement the cloud-centric approach resulting in a 3-tier architecture (i.e., cameras, edge devices, and cloud servers). Figure 1.2 shows an example deployment of IoT applications that leverage the 3-tier architecture. The deployment includes IoT devices (E.g., cars, surveillance cameras, tethered-cameras) that capture visual data and have a capability to transmit that data across a wireless/wired network. The deployment also includes *edge devices* that act as gateways to aggregate and forward IoT-captured data or to protect IoT devices from being exposed to security vulnerabilities. Such edge devices are typically few hops away from the data source, and they have non-trivial computational and network resources. The edge devices can come in various types that differ in size, location, and capabilities. They include: (1) end-devices (e.g., cars, smartphones, Raspberry Pi, and Personal computers), (2) Networking infrastructure (e.g., routers and switches), (3) Wireless Networking infrastructures (e.g., cellular towers). The type of available edge devices is typically dependent on the analytics application. In this dissertation, we do not make assumptions about the type of edge devices or its hardware capabilities (e.g., CPU/memory), the only assumption is that edge devices are located at a close physical proximity to the IoT devices. Hence, the information they collect does not have to travel nearly as far as it would under a traditional cloud architecture. The deployment in Figure 1.2 also includes traditional cloud resources. Such resources can be owned and managed by users (private clouds) or owned by a public cloud provider (e.g., AWS, Microsoft Azure,..etc).

Most of the existing video analytics [15] [16] systems for live video streams do not leverage the 3-tier architecture, and instead they focus on a 2-tier architecture where the encoded video is sent from the camera/smartphone to either a remote cloud server or a private edge server. The existing 2-tier approaches can reduce either transmission latency by utilizing an edge server, or computation latency by utilizing more powerful cloud servers, unlike 3-tier architectures which can make the appropriate tradeoff between optimizing both. In this dissertation, we build an orchestration platform for processing visual data streams across edge and cloud compute resources in 3-tier architectures. We elaborate on the challenges of building such platforms in Section 1.3.

1.2.3 Serverless Computing

A key enabling factor of deploying visual IoT analytics applications in public cloud infrastructure (e.g., Amazon Web Services) is Serverless computing [17]. Serverless computing refers to a new generation of platform-as-a-service offerings by major cloud providers. The first service offered in this category was Amazon Web Services (AWS) Lambda [18] which was first announced at the end of 2014, and experienced significant adoption in mid to late 2016. All the major cloud service providers now offer similar services, such as Google Cloud Functions, Azure Functions and IBM OpenWhisk.

In Serverless computing, the cloud provider takes responsibility for receiving client requests and responding to them, and performing task scheduling and operational monitoring. Developers need to only write the code for processing client requests. This is a significant change from the traditional paradigm in which development and operations staff have to explicitly manage their virtual machines. With the aid of serverless technology, rather than continuously-running virtual machines, developers can now deploy 'functions' that operate as event handlers, and only pay for CPU time when these functions are executing.

In order to match the increasing volume of data coming from Internet of Things (IoT) devices, AWS offers another service in its serverless computing ecosystem, called AWS Greengrass [19]. AWS Greengrass was first offered in mid 2017. The service allows processing data closer to the source where the data is generated instead of sending it across long routes to data centers or clouds. Greengrass supports running functions on edge devices (e.g., Raspberry Pi) that are controlled by the users and provide a tight integration between the user's edge device and the cloud infrastructure owned by Amazon. Users of Greengrass are charged per device rather than per function so no matter how many functions are running on the edge device, the cost is fixed. However, due to the limited compute capacity on such edge devices, the function execution might be significantly slower. A natural question that arises is which functions to place on the resource constrained edge devices in order to optimize the cost without dramatically increasing the latency. To answer this question, a clear understanding of the new pricing model and the factors affecting the price of serverless applications across the edge and the cloud is required. We address these problems and we develop cost optimization algorithms for serverless applications spanning edge and cloud in Chapter 6.

1.2.4 Privacy-preserving Machine Learning (PPML)

Deploying IoT applications on the edge/cloud infrastructure that are owned by public cloud providers (e.g., AWS) or public edge providers (e.g., VaporIO) poses security and privacy chal-

lenges because users remain skeptical about the confidentiality of their input data and whether private user data is used by cloud/edge computing providers for the providers' own economic benefit (e.g., Targeted Ads). This is now common that an attacker or an untrusted service provider wants to use data for their economic benefit so they do not damage the system but quietly leak the data.

Privacy-preserving machine learning (ML) techniques aim to allow public edge/cloud providers to perform predictions on user's data without releasing the private data in its original form. This was mainly performed by utilizing cryptographic approaches [20][21]. Such approaches are based on two main concepts: fully homomorphic encryption (FHE) and secure multiparty communication (SMC) which allow machine learning predictions to operate on encrypted data without ever decrypting it and the output of the computation is the same as if it happened on unencrypted data. Although such approaches achieve reasonable accuracy, their current performance makes them not practical for production environments.

On-device inference has also been proposed to protect data confidentiality [22][23] through processing parts of the application in the client's device and preventing the private data from leaving the client's device. However, such approaches consume a significant amount of computation and energy from the resource-constrained client devices. As an alternative, several systems [24][25] have proposed using trusted execution environments -also known as TEEs or enclaves- such as Intel Software Guard Extensions (SGX [26]) to run machine learning workloads while preserving data confidentiality. In such systems, the private data is only decrypted within the trusted execution environment which is protected from all privileged software in the system such as operating system and virtual machine monitors. However, SGX-based computation is currently performance- and memory-constrained. For example, TEE workloads cannot exploit accelerated linear algebra libraries. It also has a limited memory size (128 MB) which limits the amount of computation that can be done within one TEE and it becomes essential to delegate part of the computation to run in another hardware accelerator (e.g., GPU, CPU, or another enclave) that sits in the same or a different edge device. In this dissertation, we build an edge-cloud orchestration platform that supports the deployment, and management of applications, seamlessly across multiple enclaves that sit in different devices and we address a major challenge of optimally distribute deep neural network computation across an enclave and other devices (e.g., other enclaves or hardware accelerators). In the next section, we provide more background about the Trusted Execution Environment (TEE).

1.2.5 Trusted Execution Environment (TEE)

Trusted Execution Environment such as Software Guard Extensions (SGX) is available on Intel processors starting with Skylake. TEE provides a secure area in the processor that protects code

and data from all other privileged software on the platform. The code in the TEE is executed safely on secret data that nobody outside the TEE can have access to it including the hardware vendor (e.g., Intel). The privacy and integrity of the code/data inside the TEE is enforced by the hardware. TEEs are sometimes called enclaves, and we use both terms interchangeably in this dissertation. Intel SGX supports remote attestation of the code and data in the TEE [27]. This enables a remote user to verify the trustworthiness of the hardware and the integrity of the TEE contents (i.e., code and data). The size of the memory reserved to the TEE is limited to 128 MB. However, SGX supports *paging* in which the rarely used Enclave Page Cache (EPC) pages are evicted to the unprotected main memory, but they remain encrypted to ensure confidentiality. With paging, applications in TEE can use more than 128 MB at the expense of encrypting and decrypting the evicted pages which poses an additional efficiency challenge.

1.3 CHALLENGES AND PROBLEM DESCRIPTION

As discussed in the Section 1.2, there are several technologies that enable automated analysis of visual IoT data across the edge and the cloud. We argue that the current video analytics systems is not keeping up with such advances and they lack a holistic approach that unfolds and addresses new challenges that arise from: (1) Leveraging new technologies such as Serverless computing, and Privacy-preserving machine learning, (2) Keeping up with recent advances in machine learning, computer vision, and IoT.

To bridge this gap, we build a comprehensive framework for processing visual data streams across edge and cloud compute resources. This includes: (1) building a framework for processing visual data streams across edge and cloud compute resources, (2) developing algorithms that identify the best placement of computations across edge and cloud resources to optimize latency, bandwidth, price, and privacy, (3) Building the middleware systems that validate the effectiveness of the optimization algorithms and the ability of the framework to deploy and manage visual IoT applications across edge and cloud resources. The management and orchestration of visual IoT applications across edge and cloud resources is challenging due to the following requirements:

- **Stream-based data:** IoT data comes from various visual data sources as a continuous stream of observations, hence there exists some queuing delay for observations of the same data stream and across different data streams that share the same compute and network resources. Careful modeling of the queuing delay is crucial to estimate the end-to-end latency of the application.

- **Geo-distributed resources:** Compute resources that process IoT data are asymmetric and geo-distributed (i.e., edge devices are usually less powerful and possibly cheaper than us-

ing cloud resources). Such asymmetry requires careful modeling of the trade-off between computation and communication delays.

- **Arbitrary number of edge devices:** Each IoT application can deploy different numbers of edge devices. For example, some applications assume one edge device and one cloud, and other applications assume a hierarchy of edge devices and one cloud. Hence, the underlying system and algorithms should allow placing operators on multiple edge devices and multiple clouds.

- **Privacy-sensitive:** Applications such as smart homes or healthcare could have privacy concerns that prohibit some tasks from being executed on specific resources (e.g., public clouds). To preserve the privacy and confidentiality of the users' sensitive data, the system has to either: (1) support pinning computation tasks to on-premise edge devices, or (2) efficiently leverage the resource-constrained Trusted Execution Environments (TEE) such as Intel-SGX.

- **New public cloud paradigms and pricing models:** The presence of edge resources, owned and managed by the user, have caused cloud providers to devise new pricing models and paradigms for connecting edge devices to the cloud. Serverless computing has recently emerged as the prominent solution for analysis of IoT data in public clouds, where the analysis can be done on user-owned edge devices or public cloud infrastructure. Hence, the system that deploys distributed edge and cloud applications has to support deploying serverless applications and reason about different factors affecting their prices.

- **Application-agnostic data compression:** Each IoT data source (e.g., cameras) is designed to transmit the captured data in a compressed format to efficiently utilize bandwidth. However, data compression algorithms have been designed for a long time, aiming at increasing the compression ratio, reducing the compression/decompression time, and pleasing human viewers while being agnostic of the downstream analysis tasks (e.g., object recognition, anomaly detection, etc). However, this is bandwidth inefficient because video encoding schemes, such as H.264, might send data tailored for human perception but irrelevant for the overall analysis goal.

Problem Description: Based on the presented challenges, we conclude that visual IoT applications can be defined by four main components: (1) The application specification that describes the operations/functions to be processed on the data and the order of their execution, (2) The compute resources that will execute the operations. These resources include the IoT devices, edge devices,

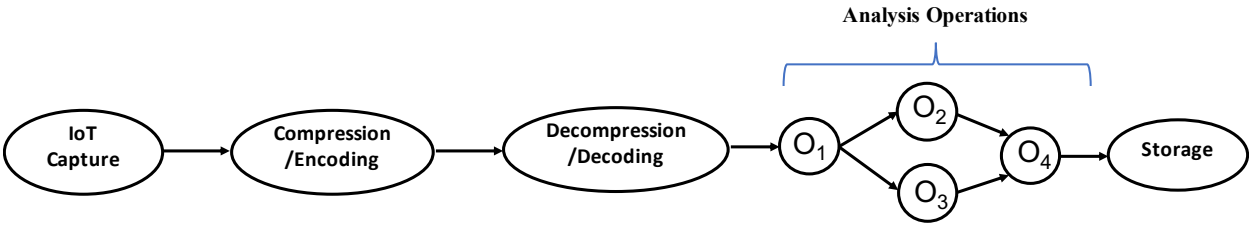

Figure 1.3: Application Graph.

and the cloud compute resources , (3) The characteristics of the data to be processed which include data type, data rate, and data size (bytes), and (4) The system utility/metric that we need to optimize and such utilities include bandwidth, latency, locality, price and privacy.

When designing distributed systems for visual IoT applications, the problems we need to solve are: (1) How to define the application specifications, compute resources, and the utility function(s) that we need to optimize, (2) What are the tradeoffs between various utility functions (e.g., does optimizing for higher privacy result in slower execution ?), (3) What are the algorithms that we can design to address such tradeoffs, and (4) What are the middleware systems we need to implement to validate the optimization to our utility function(s) (e.g., optimizing the price requires building a system that utilizes public cloud infrastructure such as AWS).

1.4 DEFINITIONS AND ASSUMPTIONS

As described in the previous section, visual IoT applications are defined by four main components: the application graph, resource graph, data model, and utility functions. We define these components as follows:

Application graph: We model the visual IoT application as a directed-acyclic-graph of operations (Fig. 1.3). The graph consists of compression/decompression operations, user-defined analysis operations, and storage operations. The application starts by compression which is typically done before transmitting the data to the edge device, the data is then decompressed where an arbitrary number of analysis operations is performed on it. After the analysis is done the result of analysis is stored in a long-term storage database/filesystem. Each analysis operation is a processing element that can execute user-defined code (e.g. convolution or face detection). Over the course of this dissertation, we assume that analysis operations can be at different granularities. For example, in chapter 6, we deal with high level operations such as (e.g., face detection). However, in chapter 7, we deal with low-level operations such as (convolution, pooling, and matrix multiplication).

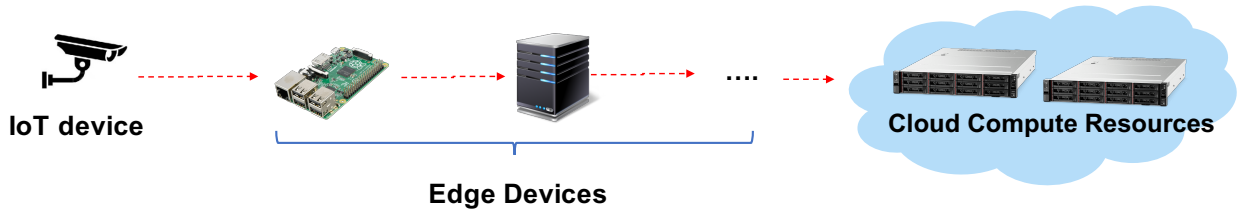

Figure 1.4: Resource Graph.

Resource graph: We model the physical resources as a weighted directed graph as shown in Figure 1.4, where the vertices represent IoT devices, edge devices, and cloud devices. The edge and cloud devices can be user-owned or public-cloud-owned and each device can have a trusted execution environment (TEE). The links between the devices represent bandwidth availability from the source device to the destination device and vice versa.

Data Model: We model the data as an unbounded stream of images (video frames) that can come from one or more camera sources as shown in Figure 1.5. We assume that the stream of frames has fixed frequency (e.g., 30 frames/sec). We aggregate a sequence of video frames into *chunks*. The number of video frames in the chunk is an application-defined parameter and we assume that the video frames within a chunk have equal sizes (e.g., 5 MB/frame).

Utility: We define a utility as the application's metric that we aim to optimize. In this dissertation, we explore various utilities including bandwidth, latency, locality, price, and privacy. We note that in many cases there is a tradeoff between these utilities. For example, in order to preserve the privacy of the data being analyzed, one might have to process the application in a secure/encrypted environment which might negatively affect the latency. An important utility is the *locality* which is defined by the ability to process the data as close as possible to where the data is generated. The application has the highest locality when the processing happens in the IoT device or the close-by edge devices. Higher locality typically results in better privacy because the data does not need to be transmitted to a third-party but it might negatively affect latency if the IoT/edge device has limited computation capacity. Our framework has the ability to control the locality (i.e., placement of operations across devices) to control the tradeoffs between various utilities. In chapters 5,6, and 7 we show how controlling the locality (placement) can affect the latency, price and privacy, respectively.

1.5 THESIS CONTRIBUTIONS

In this dissertation, we aim to address the above challenges by (1) building a framework for processing visual data streams across edge and cloud compute resources, and, (2) developing al-

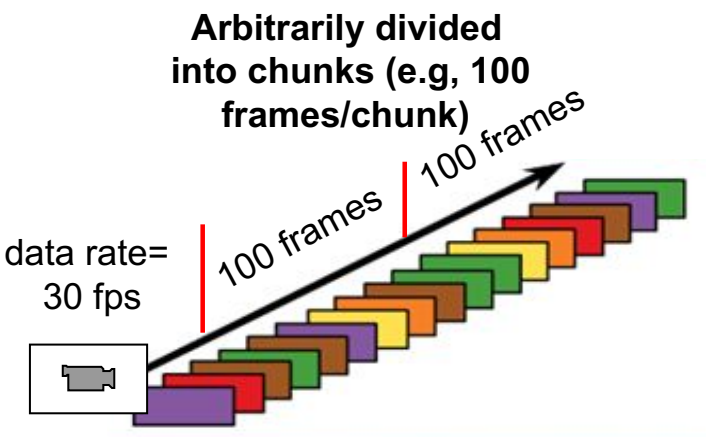

Figure 1.5: Visual IoT Data Model.

gorithms that identify the best placement of computations across edge and cloud resources to optimize bandwidth, latency, locality , price, and privacy.

The framework and the algorithms optimize various utilities and address the tradeoffs between them. The first algorithm focuses on optimizing the bandwidth by detecting the events of interest as close as possible to where the video is generated. To achieve this, we develop *Semantic Video Encoding* technique in which we redesign the video compression algorithms at the camera to be aware of the edge-based downstream analysis tasks and produce key-frames only when a semantic event happens (e.g., new object enters the scene). This allows edge-based analysis tasks to search the compressed video for the parts that are relevant to the overall analysis goals.

The second algorithm focuses on optimizing the application's end-to-end latency. To achieve this, we develop an *Operator Placement* algorithm that is given a processing job expressed in the form of a Directed Acyclic Graph (DAG) of operators/functions, it finds which operators to place on an edge device and which operators to place on a remote cloud server with the goal to optimize the overall computation and transmission latency. A key contribution in our approach that was not explored in mobile computation offloading literature is to leverage *pipeline parallelism* in which we consider that both the edge device and the cloud server are concurrently executing different video frames. This pipelining is beneficial because the interarrival time between frames is less than the processing time of one frame which allows both the edge and the cloud to be concurrently processing different frames. Our function placement algorithm scales to thousands of operations and tens of geo-distributed resources and we call it *Droplet* [28]. The name comes from the analogy with the natural phenomenon of cloud formation in which water vapor *droplets* come together to form a big cloud. Similarly, our algorithm aims to place computations on small edge devices (i.e., droplets) so that their aggregate compute capacity will be brought together to form a big cloud data center.

The third algorithm answers the question of *how to perform hybrid edge and cloud analytics in public clouds in a price-efficient manner*. To answer this question, we develop an algorithm to optimize the price of deploying analytics applications in serverless computing platforms (e.g. AWS Lambda). We identify several factors that affect the price of serverless application which include (1) *Function Fusion* in which we fuse multiple functions in a workflow to reduce the price of data transfer and state transitions from one function to another, and (2) *Function placement*, in which we explore the price-latency tradeoff between placing functions in cheap resource-constrained edge device vs placing functions in the costly resource-abundant cloud.

The fourth algorithm focuses on optimizing the end-to-end latency of computation while maintaining data privacy. To achieve this, we leverage trusted execution environments (TEE) to perform the analysis, and the data is sent securely to the TEE which decrypts the data, performs the computation, and sends the output in an encrypted form. To speed up the computation, we develop a technique to find the best partitioning of neural networks computation across multiple $TEEs$.

1.6 THESIS STATEMENT AND OUTLINE

In summary, my **thesis statement** is: *To control the tradeoff among utilities, such as bandwidth, latency, price and privacy, for visual IoT analytics applications over diverse edge-cloud enviroments, we must consider a holistic placement optimization framework integrated with semantic video encoding, function fusion and function decomposition services.*

The dissertation is organized as follows. In Chapter 2, we summarize the related work in edge-cloud analytics systems and how these systems optimize the application deployment on the edge and cloud. In the following chapters, we describe in detail different components of the dissertation. In Chapter 3, we present a high-level architecture of the dissertation's main components. In Chapter 4, we challenge the assumption that videos are compressed to be watched by humans and we show how we can optimize the bandwidth by training video compression algorithms to understand the semantics of the downstream analysis task. In Chapter 5, we describe our operator placement algorithm and we show its effectiveness in reducing the application's end-to-end latency. In Chapter 6, we describe the implications of performing the analytics on Serverless computing architecture and we discuss our price optimization algorithm for serverless computing. In chapter 7, we describe how to do the analysis in a privacy-preserving fashion by leveraging trusted execution environments. We conclude the dissertation, summarize lessons learned and future directions in Chapter 8.

CHAPTER 2: RELATED WORK

We divide the related work into four main sections. In Section 2.1 we focus on discussing the related work in edge-cloud orchestration frameworks which is a core component that we build the rest of the components upon. In section 2.2, we discuss the related work in the Visual IoT systems that are built on top of the edge-cloud orchestration framework. In section 2.3, we discuss the related work in privacy-preserving machine learning systems. In section 2.4, we discuss the algorithms and techniques used to optimize latency, bandwidth and monetary cost of such systems. For each section, we discuss various approaches and assumptions for systems that utilize cloud resources only, edge resources only, or hybrid edge and cloud resources which is the topic of this dissertation.

2.1 EDGE-CLOUD ORCHESTRATION FRAMEWORKS

The conventional approach of performing analytics on data streams is to send the streams to a centralized data center (cloud) and leverage its powerful and seemingly abundant resources to execute the analytics remotely. For the last decade, several stream processing systems have been developed to support large-scale stream analytics in the cloud such as Apache Storm [29], Apache Spark Streaming [30], Apache Flink [31], Apache Nifi [32]. However, given the tremendous amount of data transfer required by video data as an example, the latency and bandwidth requirements become significantly high. Therefore, Edge [14] and Fog [33] resources, which are typically few hops from the data source, became actively considered as additional compute resources to complement the cloud-centric model. Similar to Apache Storm in the cloud, there have been several stream-processing frameworks developed for the edge such as Edgent [34], IBM's Node-RED [35], MiNIFI [36], Eclipse Kura [37], and VMWare Liota [38].

In our work, we target a higher level abstraction in which we design and validate an edge-cloud orchestration framework that simplifies the composition, deployment, and management of tasks across edge and cloud resources. In such a platform, given a DAG of operations being executed over a stream of data, the orchestration layer will decide what operations to deploy in the cloud's stream processing engine and what operations to deploy in the edge stream processing engine. Our orchestration platform is built on top of the Echo [39] platform which is an open source orchestration platform that uses Apache Nifi engine for cloud stream processing and Apache MiNiFi engine for edge stream processing. In our platform, we also provide a novel operator placement algorithm that we compare with the related work in Section 2.4.1.

There are existing commercial solutions developed by the public cloud providers to deploy ap-

plications in a hybrid edge and cloud environment such as AWS Greengrass [19] and Microsoft Azure IoT Edge [40]. These solutions can be installed on edge servers to provide compute capability at the edge server and tight integration between application's edge servers and the cloud infrastructure owned by Amazon or Microsoft. In this dissertation, we explore the problem of optimizing the price of edge-cloud analytics in public clouds and we build our system on top of AWS Greengrass in the edge and AWS Lambda [18]/AWS Step Functions [41] in the cloud. We note that AWS Greengrass is an AWS service that has similar functionality to Apache Minifi. Sumilarly, AWS Lambda together with AWS Step functions provide similar functionality to Apache Nifi. We note that our orchestration framework can deploy functions on AWS Greengrass and AWS Lambda using their respective APIs. Similarly, our framework deplpys functions on Apache Minifi and Apache Nifi when using public cloud is not a requirement.

2.2 VISUAL IOT ANALYTICS SYSTEMS

Another layer of abstraction that this dissertation contributes to is Visual IoT analytics and machine learning inference systems. In such systems, videos are automatically analyzed to find some insights such as *the number of cars passing a certain intersection in traffic videos*, or answer questions such as *does this object exist in that video*. This analysis could, (1) be performed in real-time to monitor real-time car traffic [42], or (2) answer questions about pre-recorded videos [15] [43], or (3) do some lightweight analysis at real-time and the rest of the analysis after the video is recorded [16]. An integral part of Visual IoT analytics systems is applying pre-trained neural network (NN) to video streams. Applying NNs to video streams consists of passing individual video frames to the NN engine, one frame at a time. For example, in order to detect objects in video, we evaluate the NN repeatedly, once per frame. The most popular approach for performing video analytics on camera streams is using Convolutional Neural Networks (CNNs) [44] [6], which have largely displaced classical computer vision methods due to their high accuracy in visual analysis tasks such as object detection [45] and object classification [6]. Several techniques have been proposed to optimize the latency, bandwidth and accuracy of NN inference on videos. The techniques range from: (1) **Video aspects**: selecting the best resolution and bitrate [42][16][46], to (2) **NN aspects**: NN compression, pruning, fusion, and lower precision, and specialized models, to (3) **Hardware aspects**: Google's tensor processing unit [47] and Intel's Movidius neural computing stick [48], and other FPGA-based hardware accelerators [49].

Putting the dissertation work in perspective, we focus on two techniques that were not explored extensively in the related work [50][51][52][53]. The two techniques are *Semantic Video Encoding* and *Neural network partitioning*. In *semantic video encoding*, we redesign the video compression

algorithms to be aware of the downstream analysis tasks and we show in Chapter 4 how this idea significantly improves the bandwidth and latency, and reduces the amount of decompressed video frames. NoScope [15] tries to achieve the same goal of reducing the number of frames undergoing NN inference, however, their approach depends on computing image similarity between consecutive frames (e.g., SIFT matching [54], and mean squared error (MSE)). The fundamental novel aspect in our approach is leveraging the motion estimation in video encoders to generate I-frames when a significant motion difference exists. Contrary to existing methods, our method alleviates the need to decode the huge number of P-frames (≈96% of the video), which results in significantly faster analysis. *To the best of our knowledge, this is the first work to propose tuning video encoders to detect changes in object labels across a video.*

The second technique that we propose is *NN partitioning* in which we divide the computation of NN layers across edge and cloud resources. Existing neural network inference systems, such as Clipper [55], TensorFlow Serving [56], Rafiki [57] focus mainly on ease of deployment, where the entire neural network is considered as a black box and deployed into one Docker container. However, they miss the opportunity of leveraging hierachical clusters through deploying some layers of the neural network on the edge server near the data source and the rest of the layers in a remote cloud. On the other hand, recent systems who have proposed placement of neural network operations across hierarchy of resources such as VideoEdge [58], and Neurosurgeon [59] did not consider partitioning a neural network for a stream of video frames which requires modeling pipeline parallelism. Both systems base their partitioning/placement decisions on a single video frame. VideoEdge partitions a sequence of NN models rather than partitioning at the granularity of layers within a NN model. Neurosurgeon shares a similar objective with our system where both systems focus on reducing the end-to-end latency, but Neurosurgeon's approach does not consider the existence of a stream of frames which results in suboptimal partitioning of NN layers.

2.3 PRIVACY-PRESERVING MACHINE LEARNING SYSTEMS

Several approaches have been proposed to address the challenge of running machine learning (ML) workloads with reasonable accuracy while maintaining data confidentiality. Cryptographic-based [20] approaches are based on two main concepts: fully homomorphic encryption (FHE) and secure multiparty communication (SMC) which allows machine learning predictions to operate on encrypted data without ever decrypting it and the output of the computation is the same as if it happened on unencrypted data. Although such systems achieve reasonable accuracy, their current performance makes them not practical for production environments. On-device inference [22][23] has also been proposed to perform ML in a privacy preserving manner through processing parts

of the application in the client's device. However, on-device inference incurs high energy cost and significantly affects the lifetime of battery-operated devices. The approach that we adopt in this dissertation is to use Trusted Execution Environments (TEE) (e.g., Intel SGX) which have significant performance benefit over homomorphic encryption and does not incur the significant energy cost associated with on-device inference.

Several systems have proposed to run machine learning workloads in Intel SGX to preserve data confidentiality [24][25][60][61][21]. However, none of their work has addressed the problem of partitioning computation across multiple enclaves or highlighted the tradeoff between using multiple enclaves vs. one enclave and another processor. Occlmency [61] and Myelin[25] focus on accelerating the computation within one enclave. Occlumency proposes memory-efficient techniques for computing convolution and incrementally loading model parameters instead of loading the entire model at once. Our work benefits from such optimizations to accelerate the partial computation deployed within each individual enclave. Chiron [24] focuses on training ML models while keeping the model parameters protected from the user. Our work however focuses on inference rather than training and the model parameters are owned by the user and protected from the cloud provider. Yerbabuena [62] and Slalom [60] focus on offloading part of the computation to a colocated processor. Our work goes beyond these works by offering partitioning across multiple enclaves and exploring the privacy tradeoffs that it entails.

2.4 OPTIMIZATION ALGORITHMS FOR EDGE-CLOUD STREAM ANALYTICS

2.4.1 Operator/Function Placement

Distributed operator placement problem has been discussed in the literature in various research areas such as Distributed Stream Processing [63], Mobile Computing [14], Service composition [64], and Sensor networks [65]. In the following, we discuss the contribution of our work relative to the approaches proposed in other research areas.

Distributed Stream Processing: Different operator placement heuristics have been proposed to minimize the application end-to-end latency [63], and the inter-node traffic [66]. The above approaches are designed for distributed stream processing frameworks such as Apache Storm [29], where the resources are assumed to be homogeneous, physically co-located in the same data center, and connected with local area network. However, such assumptions do not apply to our setup, where resources are (1) heterogeneous (e.g., Raspberry Pis, laptops, and high-end servers), (2) geographically distributed and not fully connected.

Computation Offloading: The concept of Cloudlets [14] has been proposed as an additional

layer that sits between the smartphone and the cloud to help reduce latencies while offering superior non-trivial computation resources. Previous work in mobile computation offloading addresses partitioning one application between smartphone and cloud/cloudlet from the perspective of a single user and assuming infinite compute resources at the cloud [67]. The work in [68] addresses the tradeoff between computation and communication delays for dependent operations and proposes an optimal algorithm for single-user computation offloading for sequence-based operator graphs. We, however, look at more complex graphs such as DAGs. The work in [69] addresses the NP-complete operator placement problem for DAG-based operator graphs and it proposes to solve the problem by searching the combinatorial space of solutions using genetic algorithms. However, a fundamental limitation of the above approaches is that they do not address challenges such as: (1) Modeling the queuing delay introduced by having a stream of data, and (2) Modeling the pipelined execution that results from multiple observations being processed concurrently by different operators.

Service Composition: Service composition is another related problem in which service providers decide on which computing resource the service should be allocated to meet QoS (Quality of service) requirements. Previous work models service composition as multipath constrained path finding problem [64] which is an NP-hard problem and several heuristics have been proposed to solve it (such as [70]). However, similar to single user computation offloading, none of the above approaches explicitly model the queuing delay and pipelined execution that are associated with processing data streams.

2.4.2 Function Fusion

The problem of operator fusion and code generation have received attention in the database systems and high performance computing (HPC) literature because it has the potential to reduce the intermediate data between operators/functions and the number of scans on the input data. However, most of the work in these areas deals with a finite set of operators and assumes that the operator semantics are known beforehand. For example, the database community deals with relational algebra operators such as joins, aggregations, and projection [71]. On the other hand, HPC and machine learning communities deal with linear algebra operations such as matrix multiplications and factorizations. Spoof [72] and Tensorflow XLA [73] are representative operator fusion approaches in this category, and they focus on searching for patterns of operators that are known to give better performance when fused together. On the contrary, our approach depends on profiling the application and is agnostic of the application semantics which is essential, given the huge variety of applications that run on edge and cloud platforms. Moreover, our approach goes beyond function fusion and we were able to jointly model both placement and fusion solutions in

one graph which was not addressed in the previous approaches.

2.4.3 Cost Optimization of Serverless Computing Platforms

There is a significant amount of related work in optimizing the monetary cost charged by public cloud providers while meeting a service level agreement (SLA). The problem has been addressed in the context of Resource provisioning [74], Autoscaling [75][76] to handle the fluctuations in the user request rate [77], short-term on-demand vs long-term reservation plans [78], and cloud scheduling using spot instances [79][80]. There are a variety of pricing models addressed in the related work which range from long-term yearly resource reservation plan, to on-demand pay-as-you-go VM instances, to variable-priced VM instances that allows clients to bid for spare CPU-hour resources (e.g., Spot instances). However, such pricing models address the payment for virtual machine usage at a per-hour resolution (as with AWS EC2). To the best of our knowledge, we published the first work to study the factors affecting the pricing of serverless applications and to propose an algorithm to optimize the price of serverless applications. In contrast to the classical pricing models that require per virtual machine hour payment, serverless computing's pricing model charges users per function execution, and per transition from one function to another. This comes with different challenges such as function placement and function fusion. Such problems have not been addressed in the context of serverless computing. However, there exists related work in different research areas that we summarized in the previous sections. As mentioned above, the fundamental differentiating aspect in our work is the ability to jointly model both the placement and fusion solution in one graph which allows us to model the problem as the constrained shortest path and use an efficient algorithm to solve it.

CHAPTER 3: RESEARCH OVERVIEW AND APPROACHES

As motivated in Chapter 1, we aim to design a framework and build a system for analyzing visual IoT data streams where the analytics applications can span edge and cloud resources. An architectural overview of this dissertation's research is presented in Figure 3.1. The top layer is the application layer which defines an application in the form of Directed Acyclic Graph (DAG) of functions/operators. Such operators could be high-level operators such as *anomaly detection*, *object detection*, and *image captioning* operations, or low-level operators such as *tensor multiplication* and *convolution*. The granularity of the operators is decided by the application developer.

At the lowest level of the architecture is the set of geo-distributed compute resources across the edge (e.g., Raspberry pi) and the cloud (e.g., private cluster, AWS Virtual Machines). Each edge device has a local stream processing engine that handles the execution of the sub-DAG of operators that is deployed in it. The cloud has a local stream processing that handles the execution of the sub-DAG of operators in a cluster with multiple machines. There are several existing stream processing engines for both edge and cloud which are described in detail in Section 2.1. Some examples are Apache Storm [29] in the cloud and Apache MiNiFi [36] in the edge.

On top of the stream processing engine is an edge-cloud placement orchestration framework which provides an abstraction for composition, decomposition, deployment and management of operators across edge and cloud stream processing engines. The placement orchestration engine has a *Resource Manager* which carries information about which compute resources are available to deploy the operators (i.e., how many edge devices and how many cloud devices are available). The Resource Manager receives requests to dynamically register new resources and remove old ones. When a new resource is registered, the Resource Manager maintains information about whether the resource is user-owned or owned by the cloud provider, and whether the resource contains trusted execution environments (e.g., Intel-SGX) or not. This information is later used by the optimization algorithms to decide the optimal placement/fusion of operators in order to meet latency, price, and privacy constraints. The placement orchestration framework includes an *Application manager* that sends a request to the local stream processing engine of both edge and cloud to deploy the subset of the operator DAG assigned to it. Each operator in the stream processing engine can have four different implementations based on the underlying hardware. Such implementations include: (1) Intel-SGX compatible implementation, (2) CPU-compatible implementation, (3) GPU-compatible implementation, (4) AWS Lambda serverless function. The application manager chooses the operator implementation that is compatible with the underlying hardware. We note that some operators such as NN models can be implemented once and deployed either in CPU, GPU, or Intel-SGX as we describe in Chapter 7. After deploying the operators in

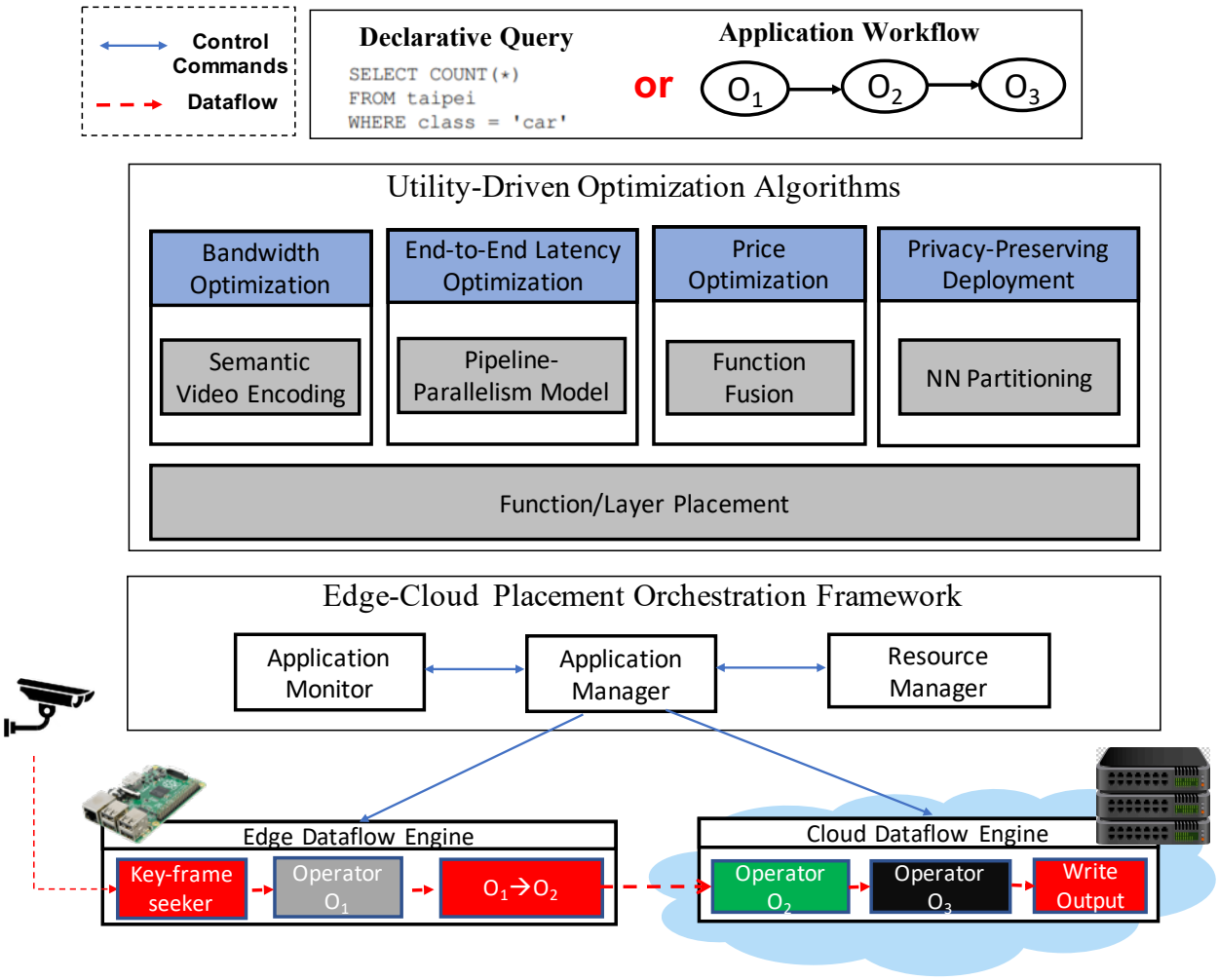

Figure 3.1: Architectural Overview of the dissertation's Research

stream processing engines, the Application Manager also allows data to flow from one stream processing engine to another by deploying a transmission operator from each stream processing engine. The transmission operator pushes the output of the final operator in one stream processing engine to the first operator in the next data-stream processing engine, hence allowing dependent operators to concurrently process video frames. When an application is deployed in the system, an *Application Monitor* keeps track of the execution times of the operators currently deployed in the edge and cloud stream processing engines. The system issues a re-placement request when the profiling information deviates from the predicted execution times.

There are several optimization algorithms that we build on top of the edge-cloud placement orchestration framework to optimize the latency, bandwidth, and price of the deployed applications. In the following we discuss the main idea of each algorithm and we summarize them in Table 3.1

3.1 BANDWIDTH OPTIMIZATION

Our bandwidth optimization algorithm relies on detecting the events of interest (e.g., a new object entering the scene) as close as possible to where the video is generated. Hence, we avoid transmitting and executing the analysis on frames that do not show significant difference from their previous counterparts. To achieve this, we present an approach where we can train video compression algorithms to be aware of the downstream analysis task and produce key-frames only when a semantic event happens (e.g., new object enters the scene). In such cases, when a video is analyzed at real-time, there is no need to decompress each frame of the video. Only key-frames are seeked and decompressed independently the same way as still JPEG images are decompressed. As shown in Figure 3.1, the key-frame seeking is performed at the edge device located close to the camera to filter the data as close as possible to where it was generated. The key-frames are passed to the downstream analysis task (e.g., Operator O_1) to undergo further analysis. Since the number of key-frames are significantly lower than the total number of frames, our technique achieves a significant improvement in bandwidth and latency by processing key-frames only, however this comes at the expense of a marginal reduction in accuracy of frame-by-frame object detection when a key-frame is not generated when a new object enters in the scene. The rest of the analysis that happens on key-frames can be performed in the edge or in the cloud based on the required latency and the available compute resources resource on each device. The placement of computation across edge and cloud is discussed in more detail in the next sections. The placement decision varies based on whether the analysis is done on user-managed devices vs cloud-provider-managed devices, and the decision also varies based on the privacy guarantees required by the users.

3.2 END-TO-END LATENCY OPTIMIZATION

Since compute resources that process IoT data are asymmetric and geo-distributed (i.e., edge devices are closer in proximity but usually less powerful than cloud resources), it becomes crucial to optimize the application's latency by controlling the tradeoff between computation and communication latencies over graph of operators processing IoT data. To achieve this, we develop an operator placement technique that receives application information from the Application Manager and resource information from the Resource Manager. Then it finds a mapping between each application operator and an available resource on which such operator is allowed to be placed. The placement service decides on the mapping of operators to resources so as to minimize the application's end-to-end completion time for a chunk of data (e.g., 1000 sensor readings or 300 video frames). The underlying placement algorithm addresses the trade-off between locality (i.e., deploying operator code near the data stream source) and latency (i.e., moving the data to a remote

	Bandwidth Optimization	**End-to-End Latency Optimization**	**Price Optimization**	**Privacy preserving deployment**
Main Utility	Bandwidth	Latency	Price	Privacy
Application Model	Compression /Decompression + one NN model	Arbitrary number of high-level or low-level operators	Arbitrary number of Lambda Functions (high-level operators)	A NN model with arbitrary number of layers (low-level operators)
Underlying Resources	Camera, Private Edge, Private Cloud	Private Edge, Private Cloud	Public Edge, Public Cloud (Serverless Computing)	Public Edge, Public Cloud (Includes TEEs)
Tradeoffs	Bandwidth, Latency, Accuracy	Latency, Locality	Latency, Locality, Price	Latency, Locality, Privacy
Approach	Semantic Encoding and placement at the edge	Pipeline parallelism / Operator placement across Edge-Cloud	Function Fusion / Function placement	NN partitioning and placement across Edge-Cloud TEEs

Table 3.1: Summary of the optimization Algorithms

cloud where computation resources are more powerful). A key contribution in our approach that was not explored in mobile computation offloading literature is to leverage *pipeline parallelism* in which we consider that both the edge device and the cloud server are concurrently executing different parts of the application graph over different video frames. For example, the first video frame that was processed by operator O_1 can now be processed by O_2, while the second video frame is being processed in parallel by O_1. This pipelining behavior is only applicable if there are enough compute resources to execute multiple operators in parallel. The pipelining behavior can reduce the computation time through parallelism, although it will introduce a transmission latency when two dependent operators are placed on different resources. Hence, the problem becomes choosing which operators to run in parallel in order to reduce the overall computation and communication time. We discuss this problem in more detail and present a solution to solve it in Chapter 5.

3.3 PRICE OPTIMIZATION

In the previous section, we discussed the approach behind operator placement to minimize the latency of processing a stream of frames, however, we assumed that the compute resources are privately owned. Many of such analytics applications are deployed on public cloud infrastructure and the price becomes an important aspect in the decision of how to place and deploy the operators. To tackle this issue, we formulate the problem of optimizing the price and execution time of serverless IoT applications. We present two models: (1) price model, and (2) execution time model that estimates the response time of the workflow of functions based on their execution and communication costs. In the price model, we identify several factors that affect the price of serverless applications which include *Function Fusion* and *Function Placement*. We present an algorithm to explore possible function fusions and placements. We represent the solutions in a structure that we refer to as the *Cost Graph* and we formulate the problem as a Constrained Shortest Path problem in which we find the solution with the best latency within a certain budget and vice versa. We discuss the problem and our algorithm in more detail in Chapter 6

3.4 PRIVACY-PRESERVING DEPLOYMENT

The main goal of our privacy-preserving deployment technique is to consider the available resources to improve the latency of the application without violating the data privacy. To achieve the maximum privacy guarantee, all the operators have to be deployed in the trusted execution environments (TEEs) that are available on either edge or cloud resources. However, due to the performance and memory constraints of TEEs (e.g., 128 MB available for Intel-SGX), our techniques tries to reduce the number of operators computed within one TEE through offloading the rest of the operators to another untrusted device (e.g., GPU) or another TEE. In this work, we focus on deep neural network computation due to their challenging memory requirements which makes it hard to run an entire neural network within one TEE. Hence, a major challenge that we address in this work is *how to partition and place the NN layers to reduce the overall latency of computation for a stream of video frames while maintaining data confidentiality*. To address this challenge we leverage an interesting insight that the output of intermediate layers of a convolutional neural network becomes dissimilar to the original input towards the last layers [81]. For example, the output of layer 5 is less similar to the original image compared to the output of layer 1. This insight can be used to run layers 1-5 inside the enclave to protect data confidentiality and offload the rest of the computation to an edge device that has a processor with more computing power (e.g., CPU or GPU). One disadvantage with this approach is that the majority of the layers might end up running in the TEE, leaving only a few layers to the faster processor. This problem is more critical in IoT

applications because such applications typically require processing a stream of video frames and if most of the layers run on the TEE, the TEE will become the bottleneck and the entire application will be slowed down by the queuing time on the enclave. Our approach to solve this is to distribute the DNN layers across multiple enclaves. This helps the partitioning to be distributed equally across both enclaves which reduces the queuing delay, provides better resource utilization, and reduces the overall latency of processing a stream of frames. We provide more details about our privacy-preserving deployment technique and our definition of privacy in Chapter 7.

CHAPTER 4: SIEVE: SEMANTICALLY ENCODED VIDEO ANALYTICS ON EDGE AND CLOUD

4.1 INTRODUCTION

Analyzing live video streams from live cameras is of considerable importance for decision making in many organizations such as traffic, police, and private security departments. A common objective of such cameras is object detection and recognition in a frame. Due to the limited computational capability of the camera devices, the conventional approach of performing object detection on camera streams is to send the streams to a centralized data center (cloud) and leverage its powerful and seemingly abundant resources to execute the analytics remotely. However, given the tremendous amount of data transfer required by video data, the latency and bandwidth requirements become significantly high (e.g., 300GB/month for Nest cameras [13]), additional compute devices known as edge devices/servers complement the cloud-centric approach resulting in a 3-tier architecture (i.e., cameras, edge devices, and cloud servers).

Utilizing 3-tier architecture poses several challenges to speed up object detection on video streams. A major challenge that we address in this chapter is to detect if the current frame has different objects than the previous frames without decoding/decompressing the full video. Addressing this challenge can significantly reduce the amount of data sent from the edge to the cloud and avoids decoding and executing expensive object detection computation on every frame of the video. Existing approaches leverage cheap image similarity computation (e.g., mean squared error) to solve this problem. They avoid transmitting frames that do not show significant difference from their previous counterparts, however, this requires unnecessarily decoding the entire video which is also a computationally intensive task.

To address this challenge, we present SiEVE [1], a 3-tier video analytics system to reduce the latency and increase the throughput of NN-based analysis over video streams. In SiEVE, we focus on object detection. Given a target video and a reference pre-trained object detection neural network (NN), SiEVE detects if a new object enters or leaves the scene without decompressing the full video. Then SiEVE uses the reference NN to detect the actual object that appears in the scene. Hence, SiEVE marginally reduces the accuracy of frame-by-frame object detection, but achieves a significant improvement in bandwidth and latency when objects do not change frequently. At the heart of SiEVE is a novel **Semantic video encoding** technique that tunes video compression algorithms to detect absence/presence of a particular object class in a video, and compress the video accordingly. This technique then alleviates the need to decompress the video for detecting

[1] SiEVE is a tool used for separating coarser from finer particles. In our system we separate coarser frames (i.e., frames that are likely to have objects) from other frames

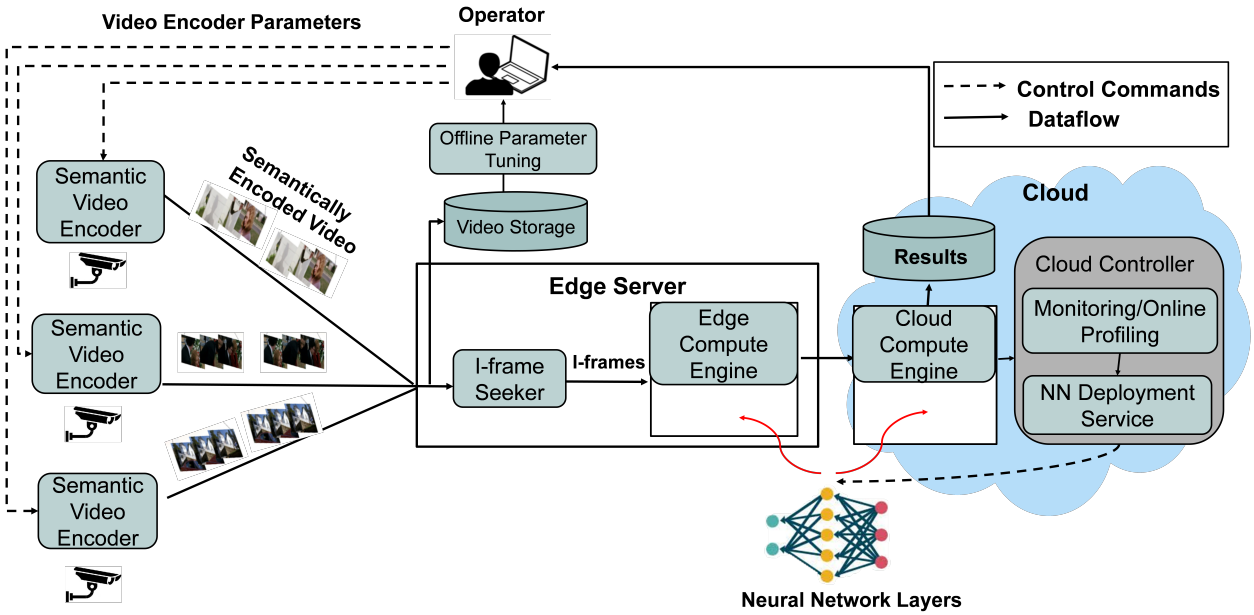

Figure 4.1: SiEVE System Architecture.

absence/presence of such objects during subsequent analysis on edge/cloud devices.

The rest of the chapter is organized as follows. We present an overview of the SiEVE's system architecture in Section 4.2. Section 4.3 describes the details of our semantic video encoder. We present our empirical results in Section 4.4. Section 4.6 concludes the chapter.

4.2 SYSTEM OVERVIEW

Figure 4.1 shows the 3-tier architecture of SiEVE. The dashed lines in Figure 4.1 show the control commands and the solid lines show the data flow. The control commands are sent by a surveillance operator who is a dedicated personnel hired by an organization (e.g., the traffic department) to control the cameras and monitor the activity happening in the system. The operator can control the parameters of the video encoder such as *GOP (Group of pictures) size*, and *scenecut threshold*. We refer to the video encoder with controllable parameters as the *S*emantic Video Encoder. The parameters of the semantic encoder are configured offline for each camera. We provide more details about the parameters and the techniques used to tune them in Section 4.3. The semantic encoder is designed to produce an I-frame (key frame) when it is more likely that this frame has objects that are different from the previous frame (e.g., a new car entered or left the scene). On the other hand, the non I-frames are likely to have the same object labels as the previous I-frame so they do not need to be analyzed separately but they get stored in the edge storage for further analysis beyond object detection (e.g., object tracking, person identification).

The edge server receives the *semantically encoded* video from the camera via a secure protocol

that the camera supports (e.g., https or rtmps). The edge server then passes the *semantically encoded* video to an *I-frame seeker* module in which only the I-frames are extracted to be processed by the downstream neural network to identify if a new object entered or an existing object left the scene. However, the non I-frames (i.e., P-frames) will not be processed by the downstream neural network and they are assigned the same object labels as the previous I-frame. We note that the I-frame seeker is not actually decoding each frame in the video but instead it searches through the video metadata and *drops* every frame that is not of type I-frame. Our empirical results show that such I-frames are no more than 3.5% of the entire video. Hence, our system saves a huge amount of computation load that is performed in the regular video decoding pipeline such as bit stream decoding, motion compensation, and Inverse Fourier Transform (IFT) for every frame.

In this section, we evaluate our method for event detection (semantic video encoder + I-frame seeker)

The frames that pass the I-frame seeker module signals an event of the presence or absence of objects in the scene. Hence, we consider the I-frame seeking on semantically encoded video as the event detection module. The frames that pass the I-frame seeker are temporarily buffered in an event queue before being dispatched by the edge compute engine. The edge compute engine is a dataflow engine that takes an I-frame as an input from the event queue, decompresses it in the same way still JPEG images are decompressed, and passes the decompressed frame through multiple layers of the neural network (NN) model for object classification. The number of NN layers deployed on the edge compute engine is decided beforehand by the *NN Deployment service*. The deployment service can choose to: (1) deploy all NN layers in either the edge or the cloud compute engine, or (2) deploy a subset of the layers in the edge engine and the rest in the cloud engine. In this chapter, we focus on the former, and we leave the details of the NN partitioning to Chapter 5. Based on the choice made by the NN deployment service, the edge compute engine computes the output of the sub-NN deployed in it and passes its output to the cloud compute engine over a secure http connection. The cloud engine computes the output of the rest of the neural networks layers deployed on it and stores the result in a database. The results are in the form of a list of tuples where each tuple consists of frame ID and the object names that appear in the frame.

4.3 SEMANTIC VIDEO ENCODER

Video compression algorithms have been designed aiming at increasing the compression ratio, reducing the encoding/decoding time, and pleasing human viewers. However, since more and more of surveillance videos are going to be watched by algorithms, we propose an approach where

we can train video compression algorithms to be aware of the downstream object detection task and produce key-frames only when a semantic event happens (e.g., new object enters the scene). In such cases, when a video is analyzed at real-time, there is no need to decompress each frame of the video. Only key frames are seeked and decompressed independently the same way as still JPEG images are decompressed. The key-frame seeking and decompression are performed at the edge device located close to the camera. The decompressed I-frames are passed to the downstream NN to identify the new object that entered or left the scene. The NN computation can be performed in the edge device or in the cloud based on the required latency and the available compute resources on each device. In Section 4.4.2, we show the end-to-end system's performance when performing the NN in the edge and in the cloud.

Video Encoder Parameters: To tune video encoders, we focus on two parameters: *scene cut threshold* and *GOP size*. We chose these two parameters because they control the number of I-frames and the duration between two I-frames. The parameters are defined as follows:

1. **scene cut threshold**: It is a threshold on the motion difference between two consecutive frames. It controls how aggressively I-frames need to be inserted. The higher the scenecut threshold value, the more sensitive it is to small motion and the more aggressive it places I-frames. Therefore, when the scenecut threshold is set to a high value (maximum 400) then more I-frames are created compared to setting the scenecut threshold to a low value (e.g., 20). The motion difference between frame F and frame $F+1$ is calculated by how much each pixel in F has changed its position in $F+1$. The total motion difference is calculated and subtracted from the maximum possible motion. If the result value is lower than the scenecut value, a "scene-cut" is detected and an I-frame is placed.

2. **GOP size:** It is the duration between two I-frames (key frames). It is essentially the number of P and B frames between two I-frames. Hence, when the GOP size is set to a high value (e.g., 1000) then less I-frames are created.

Offline Tuning: Due to the differences in camera positions and orientations, our approach focuses on tuning the encoder parameters for each camera independently. For example, we tune our parameters to find objects in the *"Jackson town square"* surveillance camera feed. To understand why we tune each camera separately, let us consider two cameras placed at the height of 5 and 10 meters from the road, respectively. The cars in the second camera will appear smaller (i.e., consume less number of pixels) than the cars in the first camera. Hence, the amount of motion (i.e., scenecut threshold) that signals a car entering the scene is smaller in the second camera compared to the first camera.

Our approach to tune the video encoder parameters (i.e., *GOP size* & *scenecut threshold*) is to leverage historic labelled data that show the event of interest. For example, we collect several hours

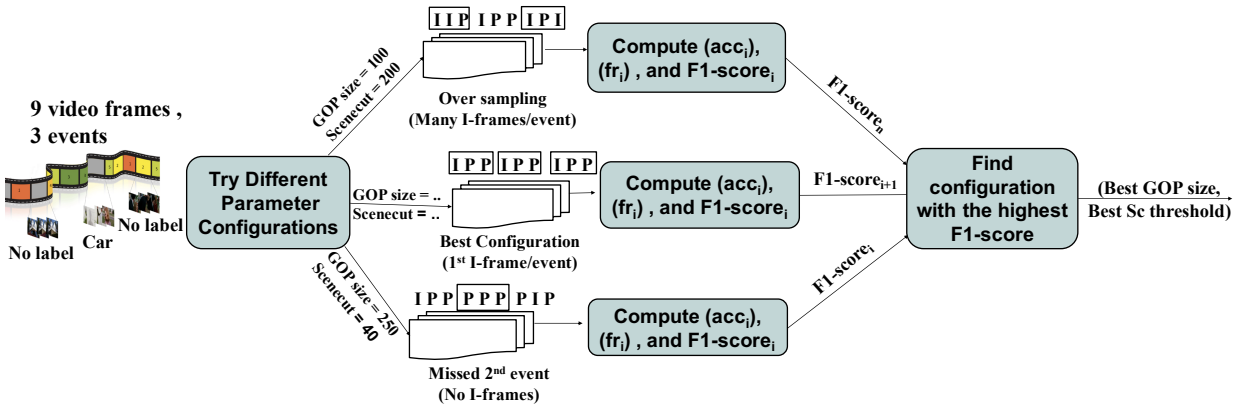

Figure 4.2: Steps of finding the best video encoder configuration for detecting object changes in compressed videos (Offline stage).

of video from a surveillance camera and we label events such as a new object entered the scene or an object that used to be in the scene is not visible any more. We can then tune the parameters of the video encoder based on the labelled events and we use the tuned parameter to detect future events in real-time.

To understand how we define events, we take an example of a 30 seconds video in which the scene has no cars for 10 seconds, then a car enters the scene and remains there for 10 seconds before it leaves the scene. We define 3 events in this video, where each event has 300 frames (10 seconds * 30 fps) and all frames within one event have the same object label. The first event has no label, the second event has the label car, and the third event has no label. We define the best event detection algorithm as the one that outputs the first frame of each event. This ensures that assigning the same object label for subsequent frames within the event will result in correct object labels.

Figure 4.2 shows the detailed steps of tuning video codec parameters. We note that these steps are performed offline to find the best video encoding parameters for a given camera feed. The best parameters are then stored in a lookup table to be used for real-time event detection. The steps are described as follows:

(1) **Step 1:** Instead of using the default parameters (i.e., GOP size = 250, and scenecut = 40), we try different configurations for the two parameters *GOP size* and *scenecut threshold* offline using historical data. We experiment with the k values for GOP size (e.g., 100,250,1000,5000) and l values for scenecut threshold (e.g., 20,40,100,200,250), so the total number of configurations is $k*l$. For each configuration, we re-encode the video with the corresponding parameter values. The result of this step is $k*l$ videos where each video has different numbers and positions of I-frames. The values of k and l define how many configurations are explored. The more configurations are explored, the more likely it is to get a better semantically encoded video (i.e., one I-frame per

event). Since this process is done offline, the values of k and l are not critical to the real-time analysis of the videos. We choose five configurations for each parameter (i.e., $k = 5$ and $l = 5$).

(2) **Step 2:** We evaluate each parameter configuration i (i.e., (GOP_i, $scenecut_i$)) by two metrics: (1) the accuracy of the event detection denoted by acc_i, and (2) the filtering rate, denoted by fr_i. The acc_i is calculated based on the positions of I-frames in the encoded video. If each event starts with an I-frame, then the accuracy is 100%. However, if an I-frame only appears in the middle of an event then the accuracy is reduced by the percentage of frames from the start of the event until this I-frame with respect to the total number of frames in the video. On the other hand, fr_i is calculated as the ratio between the number of non I-frames and the total number of frames. We note that there is a tradeoff between the acc_i and fr_i because with more I-frames the acc_i is likely to increase but the filtering rate decreases. To combine the two metrics in one quality metric, we calculate the harmonic mean (F1-score) between acc_i and fr_i. The F1-score is measured as:

$$F1score_i = \frac{2 * acc_i * fr_i}{acc_i + fr_i} \tag{4.1}$$

(3) **Step 3:** We choose the configuration that has the highest F1-score: $i^* = \underset{i}{\operatorname{argmax}}\left(F1score_i\right)$

The configuration with the highest F1-score balances the tradeoff between trying to filter as much redundant information as possible and getting a high event detection accuracy.

Online Usage of Tuned Parameters: The best encoding parameters for each camera are stored in a lookup table . The parameters are entered by the system's operator in the software provided by the camera's vendor as shown in Figure 4.1. The new parameters will then be used for real-time encoding of future live videos. The semantically *encoded* live video including I and P frames is then received at real-time by the I-frame seeker module (Figure 4.1) that sits on the edge device which in turn searches for I-frames and sends them to the event queue for further processing by a neural network.

Use cases: The current prototype of the semantic encoder focuses on detecting the existence of new objects in surveillance cameras. It has its best results when the camera has a fixed-angle and the objects entering the scene create significant motion differences. We use that technique to detect the object labels in each frame without decompressing the majority of the frames. The object labels for each frame can then be used to do further analysis such as object tracking and person identification. The semantically encoded video that we store in the edge helps to quickly seek the exact event/GOP that can be further analyzed which significantly speeds up the analysis. A limitation in our approach is that we assume that the edge location has access to non-trivial storage capacity. We also note that several cameras have hardware encoders built into them with limited control over their parameters. In these cases, we re-encode the video with the semantic

Dataset name	Object	Resolution	FPS	Duration	Description	labels ?
Jackson square [82]	car, bus, truck	600x400	30	8 hours	vehicles going back and forth in a public square	Yes
Coral reef [82]	person	1280x720	30	8 hours	people watching coral reefs in an aquarium	Yes
Venice [82]	boat	1920x1080	30	8 hours	boats moving in the lagoon	Yes
Taipei [83]	car, person	1920x1080	30	4 hours	vehicles and people in a public square in Taipei	No
Amsterdam [84]	car, person	1280x720	30	4 hours	Road intersections in amsterdam	No

Table 4.1: Datasets used in the evaluation.

parameters on the edge device.

4.4 EVALUATION

System setup: The computing infrastructure consists of edge and cloud resources. We use one desktop as the edge device and one server as the cloud. The edge device has Intel Core i7-5600 CPU with 12 GB of memory and the cloud server has Intel Xeon E5-1603 CPU with 32 GB of memory. We control the bandwidth from edge to cloud server to be 30 Mbps which simulates an average wide area network connection. Each of the edge and cloud servers has a local dataflow engine, Apache NiFi, that handles execution of operators that are deployed on it. Nifi is an engine designed for composing user-defined operators and executing dataflows in a single machine or across multiple machines. Each of the edge and cloud servers has a local deployment of Nifi and we use Echo [39] orchestration framework to handle the communication between the two Nifi instances.

Datasets: We experiment with five different datasets. The five datasets vary in the object types (car, bus, truck, person, boat), resolutions (400p, 720p, 1080p), and locations (indoor, outdoor, different cities) as shown in Table 4.1. The first three datasets have publicly available ground truth object labels. We experiment with 8 hours for each of them. The first 4 hours are used as a training set to tune the encoder parameters for our approach and the thresholds for the compared approaches. We use the next 4 hours for evaluation. The last two datasets are obtained from YouTube live feeds and they are used to evaluate the end-to-end system performance.

In the following, we conduct experiments to evaluate the improvement of the event detection module separately. We then present results for the improvement in the end-to-end performance of the system.

4.4.1 Evaluation of Event Detection

Metrics: In this section, we evaluate our method for event detection (semantic video encoder + I-frame seeker) with other approaches. We use the following metrics for comparison:

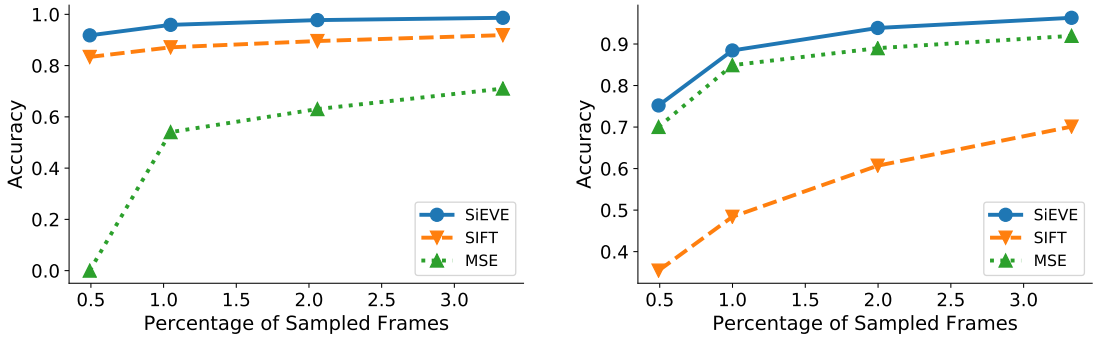

Figure 4.3: Accuracy at different sampling rates for Jackson sq. dataset (left) and Coral Reef dataset 2 (right)

(1) **Accuracy of per-frame object detection**: measured by percentage of frames with correct object labels with respect to the total number of frames.

(2) **Percentage of sampled frames (SS)**: The percentage of frames that pass the I-frame seeker and undergo the NN processing with respect to the total number of video frames.

(3) **Speed of execution**: measured by the number of frames per seconds (fps) that can be processed by the event detection module.

Compared Approaches: we compare our approach with two other approaches. The two approaches rely on decoding each frame and computing an image similarity metric between the current frame and the previous frame. If the similarity is below a certain threshold, an event is detected. The frames before the next event are assigned the same object labels as the previous frame. The approaches we evaluate are:

(1) **Mean squared error (MSE):** pixel-by-pixel mean squared difference between consecutive frames.

(2) **SIFT feature matching:** SIFT features are computed for each decoded frame and matched with the previous frame.

Accuracy: We compare between the approaches based on the accuracy of per-frame object labels at different sampling rates. For example, we try different configurations of GOP size and scenecut threshold for SiEVE. Each of them gives a different number/position of I-frames and hence different accuracy. We show the accuracy of per-frame object detection when the number of I-frames is between 0.5% and 3.5% of the entire video stream. We tune the thresholds for other approaches to give the same sampling rate as SiEVE and we compare between the approaches in terms of accuracy at each sampling percentage. We present the results for the three datasets that we have ground truth labels for. We show the results for the first two datasets in Figure 4.3. Due to space limitations, we omit the figure for the third dataset and we describe the summary of the results. The results show that for the three datasets SiEVE can achieve more than 95%

Dataset	Semantic			Default		
	Acc	SS	F1	Acc	SS	F1
Jackson sq.	98.3%	2.1%	98.1%	72.6%	0.72%	83.9%
Coral reef	99.1%	2.8%	98.16%	67.8%	0.75%	80.7%
Venice	96.5%	1.1%	97.6%	83.8%	0.4%	91%

Table 4.2: Comparison between semantic and default parameters in terms of accuracy (Acc), sample size (SS) and F1

accuracy with analyzing 3.5% of the video frames. SiEVE outperforms the related approaches by a significant margin in the three datasets. For the first dataset, SiEVE outperforms SIFT and MSE by an average of 11% and 48%, respectively. In the second dataset, SiEVE outperforms SIFT by 35% and MSE by 8%, and in the third dataset SiEVE outperforms SIFT by 28% and MSE by 7%. An interesting observation in the second and third datasets is that contrary to the first dataset, MSE outperforms SIFT. This is due to the different objects that are being detected in each dataset. MSE is well suited for detecting small objects (e.g., person, boat from long view) entering and leaving the scene which is the case in the second and third datasets. However, SIFT performs better for bigger objects (e.g., cars in close-up view) that cause significant changes in the scene. Our approach benefits from tuning the scenecut threshold to detect bigger or small objects. If a video has multiple labelled objects, the estimated scenecut threshold tends to be tuned towards detecting the object that appears smaller in front of the camera. A smaller scene cut threshold is guaranteed to detect the existence of bigger objects as well because they create more motion.

Semantic Encoding vs Default Encoding Parameters: We notice that the semantic encoding parameters that produce the best F1-score are different based on the nature of each video and are different from the default parameters (i.e., *sc=40, GOP-250*). This justifies why we tune the parameters separately for each camera feed. For the *sc* threshold, the tuned values are 100, 200, and 250, for the first 3 videos where *sc* value of 250 is more sensitive to small motion than 100. The values are consistent with the relative sizes of the objects in front of the camera. For example, the first two videos have a close-up scene on the vehicles and people so they appear bigger in front of the camera and they create more motion compared to boats in the third video that was shot from a long distance. On the other hand, the *GOP* sizes are 500, 100, 1000, which are also related to how frequent objects appear in the video. For example, the people in the aquarium appear more frequently than the boats. We show a comparison between semantic encoding parameters in terms of accuracy, sample size (SS), and F1-score in Table 4.2.

Speed of Execution: The most significant improvement in our approach lies in the speed of performing event detection which is 100-124x faster than the closest image similarity approach. We show the results for the three datasets in Table 4.3. SiEVE performs a lightweight computation in which it seeks the I-frames within a video which takes only 0.43 ms/frame (2300 fps) for 1080p

Dataset	SiEVE	MSE	SIFT
Jackson square	19600	157	115
Coral reef	7200	62	38
Venice	2300	22	16

Table 4.3: Speed of event detection results in terms of how many frames can be processed per second (fps)

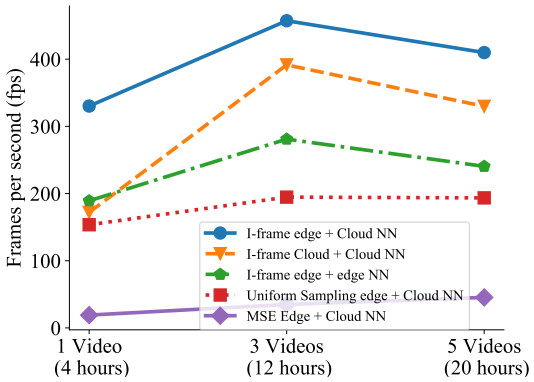

Figure 4.4: Number of processed frames per second by different baselines.

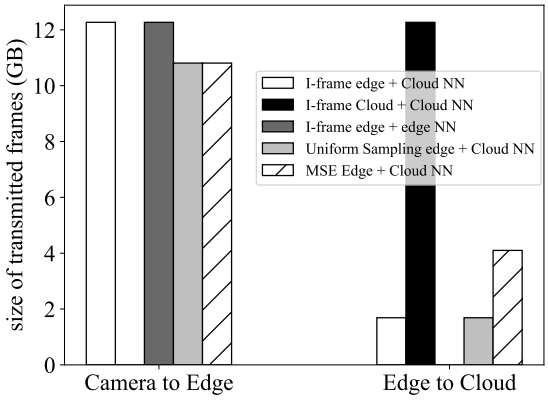

Figure 4.5: Total amount of data transfer for different baselines.

frame resolution (dataset 3). On the other hand, the other approaches are bounded by time for decoding each video frame which takes 8 ms/frame (120 fps) for the same frame resolution. In addition to frame decoding, computing image features and image similarity drives the speed down to 22 fps for MSE and 16 fps for SIFT which results in 104x and 142x slowdown compared to SiEVE. We notice that the same gain is carried over to the small resolution of 600x400 pixels. The speedup of SiEVE is 124x over MSE and 170x over SIFT.

4.4.2 Evaluation of End-to-End System

In this section, we evaluate the end-to-end performance of the system with respect to the system's throughput and the amount of data transmission from the edge to the cloud. We evaluate the throughput in the post-event analysis scenario in which the semantically encoded videos are pre-recorded and stored in the edge server and we use the five videos specified in Table 4.1. We use 4 hours from each video with the total of 20 hours. We compare our method with the following baselines:

(1) **I-frame edge + cloud NN**: I-frame seeking in the edge, and NN inference in the cloud.

(2) **I-frame edge + edge NN**: I-frame seeking in the edge and NN inference in the edge.

(3) **I-frame cloud + cloud NN**: full video is streamed to the cloud where both I-frame seeking

and NN inference are performed.

(4) **Uniform Sampling:** This approach includes uniformly sampling frames in the edge at fixed intervals, and transmitting the first frame in each interval for NN inference in the cloud. For fair comparison, we set the interval such that the number of transmitted frames is equal to the number of I-frames transmitted by the previous baselines.

(5) **MSE Edge + Cloud NN:** This approach includes executing MSE at the edge, and transmitting only the frames that pass a certain threshold to the cloud for inference.

The first three methods that implement I-frame seeking operate on semantically encoded videos while the other two methods operate on the video with the default encoding parameters. We chose the threshold of *MSE* and the semantic encoding parameters that achieves an F1-score of 95% in the training set. For the two videos that we do not have ground truth labels we set the I-frame rate to 1 frame per 5 seconds for both approaches. The total number of frames in the 5 videos including I and P frames is 2.16 millions. Figure 4.4 shows the throughput results in terms of the number of frames per seconds (fps) (i.e., $total\ number\ of\ frames\ /\ total\ time\ in\ sec\ to\ process\ all\ frames$). From the results, we observe two important insights: (1) The first 3 methods that require semantic encoding significantly outperform the other two baselines including the lightweight uniform sampling. The reason is that uniform sampling requires decoding a large amount of P-frames unlike semantic encoding which focuses on I-frames only. (2) We notice that the 3-tier architecture (i.e., camera, edge, cloud) leveraged in the first approach achieves significant speedup compared to the 2-tier architecture leveraged in the second two approaches (i.e., cloud only or edge only) because the 3-tier architecture benefits from the data filtering at the edge and the fast NN inference at the cloud.

We show the results for the amount of data filtering in the edge in Figure 4.5. The figure shows the amount of data transfer from camera to edge and from edge to cloud. We use YoloV3 as the NN inference model. From our experiments, we note that one of the limitations of semantically encoded videos is that they tend to have more I-frames than the original video. Hence, the data transmitted from the camera to the edge is 12% larger than the original video. However, after extracting I-frames and resizing them to the resolution of the YOLO model (i.e., 300x300), the size of the transmitted data from the edge to the cloud is reduced by a factor 7 (12.26GB to 1.688GB for I-frame edge + NN Cloud). Moreover, we note that the size of data transmitted by MSE is 2.5x larger than the aggregate size of I-frames which makes semantic encoding and I-frame seeking a more bandwidth-efficient approach.

4.5 DISCUSSION

The variety of types, locations, and capabilities of surveillance cameras poses an interesting question about how Sieve addresses such variety. In the following we discuss how Sieve addresses the variety in each category.

Camera type: Our framework is not restricted to using cameras with certain image/video resolution (e.g., 720p, 1080p). The parameters, that we tune in Sieve, control the number of I-frames and the duration between two I-frames. The tuning procedure works the same way regardless of the resolution of the I-frame. The same applies to the I-frame seeking procedure which is concerned with the frame type (e.g., I-frame or P-frame) rather than the resolution/byte size of the frame. Regardless of the camera's image/video resolution (quality), we assume that the camera uses h264 or h265 as the default video encoder and that it provides the capability to update the default values of the encoder's parameters such as scenecut and GOP size.

Camera location/orientation: When dealing with multiple cameras, placed at different orientations and distance from the objects of interest, our framework needs to tune the parameters (i.e., GOP size, scenecut). To understand why we tune each camera separately, let us consider two cameras placed at the height of 5 and 10 meters from the road, respectively. The cars in the second camera will appear smaller (i.e., consume less number of pixels) than the cars in the first camera. Hence, the amount of motion (i.e., scenecut threshold) that signals a car entering the scene is smaller in the second camera compared to the first camera.

Camera Stream Encryption: We note that some cameras come with security/privacy features such as transmitting the camera stream over an encrypted channel such as HTTPs or RTMPs to prevent third parties from intercepting the camera live stream. In order to analyze such encrypted streams, we discuss in Chapter 7 a privacy preserving analysis approach using hardware enclaves in which the encrypted video stream is only decrypted and analyzed in the trusted hardware enclave. We present more details about our privacy preserving approach and the analysis we perform on the video stream in the trusted enclave in Chapter 7.

4.6 CONCLUSION

In this Chapter, we present SiEVE, a 3-tier video analytics system to reduce the latency and increase the throughput of NN-based analysis over video streams. In SiEVE, we address the problem of semantic video encoding in which the video encoder becomes aware of the downstream object detection task. We show that video encoders can produce I-frames when an object enters or leaves the scenes. This allows the video to be analyzed through seeking I-frames only rather than decoding the entire video which results in 100x speedup.

CHAPTER 5: DROPLET: EDGE-CLOUD ORCHESTRATION AND OPERATOR PLACEMENT FRAMEWORK

Visual IoT applications consists of a set of dependent operations modeled as dataflow graphs. Each operation describes some computation on the incoming data such as convolution, encryption, or filtering. A key challenge that we introduced briefly in Chapter 4 is to automatically decide how to partition such operations among edge and cloud compute resources, in order to minimize the overall completion time of the entire graph of operations. We refer to this problem as *distributed operator placement* and we address it in more detail in this chapter.

The unique nature of IoT applications poses several challenges for distributed operator placement problem because IoT applications have the following properties: (a) Data comes from various sources as continuous streams of observations. Hence, there exists queuing delay for observations within the same data stream and across different data streams that share the same compute and network resources; (b) Compute resources are heterogeneous and geo-distributed (i.e., edge devices are more resource-constrained but closer to the data source). Such resource asymmetry requires careful modeling of the trade-off between computation and communication delays; (c) dataflow operations are interdependent and the placement algorithm has to optimize their execution without violating the dependency constraint.

In this chapter, we address the aforementioned challenges in the distributed operator placement problem. We devise a general model for estimating the execution time of dataflow graphs that operate on data streams of observations. The model considers computation, communication and queuing delays, and captures pipeline parallelism (due to multiple inputs being processed concurrently by different operators in the graph).Moreover, we design a dynamic programming algorithm called DROPLET to solve the operator placement in log-linear time with respect to the number of operators. The algorithm minimizes the completion time of executing dataflow graphs on edge and cloud resources.

The rest of the chapter is structured as follows. In Section 5.1, we present an overview of the optimization goals and the tradeoffs that we address in this chapter. In Section 5.2, we formally define the system models and the operator placement problem. In Section 5.3, we present an algorithm to solve it. In Section 5.4, we evaluate our algorithm and we compare it with the closest approaches in the literature. Finally, Section 5.5 concludes the chapter.

5.1 OPTIMIZATION GOALS

In this section we present an overview of: (1) The utility that we aim to optimize, (2) the applications that we addres, (3) the underlying compute resources that we levarage, and (4) the tradeoffs

that we explore:
Utility: We optimize the end-to-end latency (Completion time) of processing a stream of data by an analytics application.
Application: A sequence/graph of analysis operations which include high-level operations (e.g., face detection) or low-level operations (e.g., matrix addition)
Compute resources: Edge devices (closer to the data source), and remote cloud resources.
Tradeoffs: We aim to control the tradeoff between the application's end-to-end latency, and the application's locality (i.e., the ability to process the entire application closer to where it is generated).

5.2 MODELS AND PROBLEM DEFINITION

We start by describing the resource model, IoT data model, and the application Model. Then we formally define the problem. In Section 5.3, we present the intuition behind our approach and the algorithm to solve it. In both sections, we follow the notation in Table 5.1.
Resource Model: We model the physical resources as a weighted directed graph $G_R = (V_R, E_R)$ as shown in Figure 5.2, where the vertices $V_R = \{E, C\}$ represent two compute resources, one edge device E and one cloud server C. Note that in this chapter, we consider systems of two resources edge and cloud. The links $E_R = \{(B_{E,C}, B_{C,E})\}$ represent bandwidths from edge to cloud and from cloud to edge. An example resource graph is given in Figure 5.2. Each physical resource can have multiple virtual resources that can use a portion of the physical resources' CPU and memory. Virtual resources can be thought of as virtual machines (VMs), or containers. For simplicity, we refer to virtual resources as containers in this chapter. Each of E and C has maximum number of containers denoted as m_E, m_C respectively. We assume that all containers in the same physical resource are homogeneous in terms of compute and memory resources reserved for them.
IoT Data Model: We model the IoT data as an unbounded stream S of *data frames*, $s = \langle d_1, d_2, \cdots \rangle$. A *data frame* can be seen as one unit of data or a measurement defined by the application (e.g. tuple, video frame, sensor reading). The IoT application is typically defined for a stream/sequence of such data frames with fixed frequency f (e.g., 30 frames/s). We aggregate a sequence of data frames into *chunks* where each chunk has duration T. The k^{th} chunk ch_k can be defined as $ch_k = \langle d_1^k, d_2^k, \ldots d_I^k \rangle$, where $I = f \cdot T$ is the chunk size. In the rest of the chapter we deal with chunks of data. The chunk duration/size is an application defined parameter.
Operator Graph Model: We model the dataflow in an IoT application as a directed-acyclic-graph (DAG) $G_o = (V_o, E_o)$ of operators (Fig. 5.1). An operator is a processing element that can execute user-defined code (e.g. convolution or face detection in face recognition applications).

E	=	Edge compute resource (device)
C	=	Cloud compute resource
$B_{E,C}$	=	Bandwidth (bytes/sec) bet. E and C
G_R	=	Compute resources graph
V_R	=	Set of Compute resources (e.g., E, C, etc.)
E_R	=	Set of bandwidths bet. compute resources
n	=	Number of operators in operator graph
d_j^k	=	data frame j in a chunk k
ch_k	=	k_{th} chunk in a continous stream of data frames
f	=	data frame rate (frame/sec)
T	=	chunk duration (sec)
I	=	Number of frames in a chunk (chunk size)
o_i	=	Operator o_i
G_o	=	Operator graph
V_o	=	Set of operators
E_o	=	Set of dependency relationships bet. operators
$e_{i,E}$	=	Execution time of o_i in E
$e_{i,C}$	=	Execution time of o_i in C
D_{o_i}	=	Size (bytes) of output data of operator o_i
$tr(E \xrightarrow{D_{o_i}} C)$	=	Transmission time (sec) bet. E and C
L	=	Total number of compute resources
V_p	=	Set of vertices of the placement graph
E_p	=	Set of links of the placement graph
$t(d_j^k, o_i)$	=	Completion time of processing frame d_j^k after being processed by o_i.
$T(ch_k)$	=	Completion time of processing all data frames in chunk k
M	=	Mapping from operators to resources
$T(ch_k, M)$	=	Completion time of processing all data frames in chunk k according to the mapping M
$1_M(o_i, r_l)$	=	1: if operator o_i is placed on resource r_l, 0: otherwise
$o_i@E$	=	a symbol denoting the placement of o_i in E
$o_i@C$	=	a symbol denoting the placement of o_i in C
$(o_i@E \mid\mid o_j@C)$	=	a symbol denoting concurrent execution of o_i on E and o_j on C
$(o_i@E);(o_j@E)$	=	a symbol denoting sequential execution where o_i has to finish execution in E before o_j starts executing in E

Table 5.1: Table of Notation

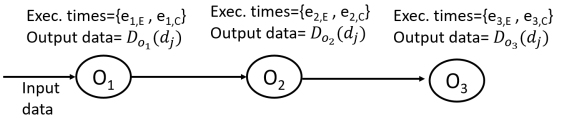

Figure 5.1: Example operator DAG.

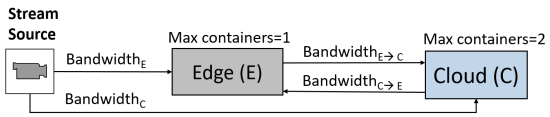

Figure 5.2: Example resource graph.

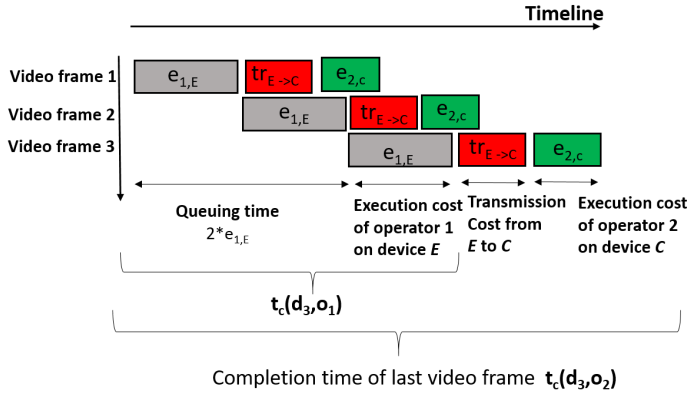

Figure 5.3: Pipelined Execution of operator o_1 on resource E and o_2 on resource C.

The vertices in the DAG represent operators $V_o = \{o_i \mid i = 1 \ldots n\}$, and the links between them $E_o = \{o_i \rightarrow o_j \mid i \neq j \ \wedge \ 1 \leq i, j \leq n\}$ represent the dataflow dependencies, where $o_i \rightarrow o_j$ means o_i is applied to a data frame before o_j is applied to the same frame.

Operator Profile: Each operator in Figure 5.1 is associated with a *profile* which includes:

1. The cost of executing operator o_i on a data frame d_j, when placed on each resource $r_l \in V_R$. Operator o_i can be placed on resource E or C so we denote their corresponding execution costs as $e_{i,E}$ and $e_{i,C}$.

   We assume that the data frames within chunk ch_k have equal sizes therefore the execution cost is the same for all data frames.

2. The size of output data of operator o_i. We denote it as D_{o_i} bytes.

3. The transmission time of the result of o_i from the edge to the cloud, $tr(E \xrightarrow{D_{o_i}} C) = \frac{D_{o_i}}{B_{E,C}}$, where $B_{E,C}$ is the bandwidth (bytes/sec) from the edge to the cloud. Since we assume that the data frames within a chunk have equal sizes, the output data of o_i is the same for all data frames i.e. $\forall d_j^k \in ch_k \ D_{o_i} = D_{o_i}(d_j^k)$.

Execution model: One of our key contributions is the ability to model concurrent execution of multiple dependent operators in a pipelining fashion. Consider the example in Fig. 5.3. A chunk of 3 video frames is processed by two operators o_1 and o_2. o_1 is placed on resource E and o_2 on resource C. First o_1 is applied to the first frame and the output is transmitted to resource C where

operator o_2 can be applied, while o_1 is concurrently applied to the second frame. Assuming $e_{1,E}$ is a large value, the time for processing the entire chunk ($T(ch_k)$) is equivalent to $t(d_j^k, o_i)$, which is the completion time of the last data frame d_j^k in ch_k and o_i is the last operator applied to it. Hence, for the above example, $T(ch_k)$ is:

$$t(d_3,o_2) = \underbrace{t(d_3,o_1)}_{\text{prev. operator}} + \underbrace{0}_{\text{queuing}} + \underbrace{tr(E \xrightarrow{D_{o_1}} C)}_{\text{transmission}} + \underbrace{e_{2,C}}_{\text{execution}} \tag{5.1}$$

$$t(d_3,o_1) = \underbrace{0}_{\text{prev. operator}} + \underbrace{2 \cdot e_{1,E}}_{\text{queuing}} + \underbrace{0}_{\text{transmission}} + \underbrace{e_{1,E}}_{\text{execution}} \tag{5.2}$$

Problem Definition: Let M be a map from operators to resources defined as $M : V_o \rightarrow V_R, M(o_i) = r_j \mid r_j \in \{E,C\}$. And let $T(ch_k,M)$ be the completion time of processing an entire chunk ch_k given the mapping M. The operator placement problem is to find a map M for which $T(ch_k,M)$ is minimized.

Note that the chunk completion time $T(ch_k,M)$ can be defined in terms of the completion time of last data frame d_I in ch_k when processed by the final operator o_n as:

$$\begin{aligned} T(ch_k,M) = t(d_I,o_n) = & \underbrace{t(d_I,o_{n-1})}_{\text{prev. operator completion}} + \underbrace{t_{queue}(d_I,o_n)}_{\text{queuing time}} + \\ & + 1_M(o_{n-1},r_k) \cdot \underbrace{1_M(o_n,r_l) \cdot tr(k \xrightarrow{D_{o_{n-1}}} l)}_{\text{transmission cost}} + \underbrace{1_M(o_n,r_l) \cdot e_{n,l}}_{\text{execution cost}} \end{aligned} \tag{5.3}$$

where

$$\begin{aligned} 1_M(o_i,r_l) &\stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } o_i \text{ is placed on resource } r_l \\ 0 & \text{otherwise} \end{cases} \\ t_{queue}(d_j,o_i) &\stackrel{\text{def}}{=} \text{queuing time of frame } d_j \text{ on operator } o_i \end{aligned} \tag{5.4}$$

Here, $t(d_I,o_0) = 0$ is the base case for Eq.5.3 and it is the time just before first operator o_1 starts.

The major challenge to solve the operator placement problem is that $T(ch_k,M)$ (Eq. 5.3) depends on the mapping M of all the operators that are running concurrently in a pipelined fashion. Hence, an exhaustive search of the solution space is required to determine the optimal values of M (in $O(2^n)$ time). There have been approaches proposed, such as branch-and-bound [85], to decrease the search complexity. But in the worst case, it is still exponential in n.

Further, DAG scheduling in general is NP-complete [86], except for 3 restrictive cases that our problem does not fall under. We also show in [87] that a previous DAG scheduling problem ([88]) can be transformed into ours in polynomial time and hence shows that our optimization problem

is NP-complete. In the following we propose an approximate algorithm to solve the problem in polynomial time with respect to the number of operators.

5.3 APPROACH

The key idea in our approach lies in the fact that due to the limited number of resources, the number of operators that can be concurrently processing data frames in a pipeline fashion is limited. For example, if we have a DAG of 10 operators executing on two physical resources, each having two containers, then we can assume that up to 4 operators can be concurrently executing in parallel and the rest of the operators will wait until at least one of the first 4 operators finishes processing a chunk and releases the container.

We note that, it may seem like fully utilizing all the available containers by running 4 operators concurrently can minimize the chunk completion time. However, doing so introduces transmission cost when two dependent operators are placed on containers that are running on different resources. Hence, depending on trade-off between execution and transmission costs, we might end up with a situation that is ideal to execute the 10 operators on only two containers that are collocated on the same physical resource. Based on the above intuition, our problem becomes choosing the operators that can run concurrently, so that the end-to-end computation and communication times are reduced as will be illustrated in the following example.

5.3.1 Illustrative Example

As shown in Figures 5.1 & 5.2, we have a sequence of three operators o_1, o_2, o_3 belonging to the dataflow graph G and two different resources: (1) one edge device E with one container, and (2) one cloud node C with two containers. We assume that intermediate output data can flow from E to C or vice versa. In this case, pipelining can happen for at most three operators which is equivalent to the total number of containers in the system.

Each possible solution to the operator placement problem requires mapping the three operators to E or C, and up to two operators can be mapped to C (many-to-one) since it has two containers. For each placement decision however, we can have different subsets of operators running concurrently in pipeline fashion. We represent the combinatorial possibilities of placing the 3 operators across E and C as shown in Figure 5.4. Each row in the figure is a possible placement solution. The label $o_1@E$ means that o_1 is placed on device E (i.e., $M(o_1) = E$), while the label $o_1@C$ means that o_1 is placed on device C (i.e., $M(o_1) = C$). The operators inside parentheses show the operators that are being executed concurrently in a pipelined manner as shown in Figure 5.3. More

1. (o_1@E) ; (o_2@E) ; (o_3@E)
2. (o_1@C) ; (o_2@C) ; (o_3@C)
3. (o_1@E) ; (o_2@E || o_3@C)
4. (o_1@C) ; (o_2@C || o_3@C)
5. (o_1@C) ; (o_2@C || o_3@E)
6. (o_1@E || o_2@C) ; (o_3@E)
7. (o_1@E || o_2@C) ; (o_3@C)
8. (o_1@C || o_2@C) ; (o_3@C)
9. (o_1@C || o_2@E) ; (o_3@E)
10. (o_1@C || o_2@E) ; (o_3@C)
11. (o_1@E || o_2@C || o_3@C)
12. (o_1@C || o_2@E || o_3@C)
13. (o_1@C || o_2@C || o_3@E)

Figure 5.4: Different possibilities for operator placement

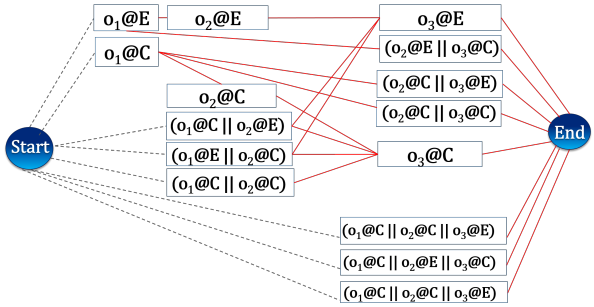

Figure 5.5: Placement graph corresponding to Fig. 5.4

precisely, the parentheses denote how a given chunk is executed and we describe the two general cases:

- **Pipelined Execution:** if we have a parentheses with two or more operators such as $(o_1@E \ || \ o_2@C)$, then the operators are running in different containers and the execution goes as follows: (1) d_1^k gets processed by o_1 in resource E, (2) once d_1^k finishes execution, d_2^k starts being processed by o_1 in resource E while d_1^k is being processed by the next operator o_2 in resource C. Hence, o_1 and o_2 keep processing operators in pipeline fashion until o_2 processes d_I^k.

- **Sequential Execution:** If the parentheses have only one operator such as $(o_1@E)$, then o_1 processes all the data frames in the chunk. The occurrence of an operator in the next parentheses separated by ';', such as $(o_1@E);(o_2@E)$, indicates that o_2 starts execution after o_1 processes the entire chunk. This behavior typically happens when the second operator o_2 is placed on the same resource E and the resource has one container. Hence, o_2 has to wait for o_1 to finish execution.

Since we have a total of 3 containers across the edge device and cloud, each parentheses can have at most 3 operators that can run concurrently.

The representation in Figure 5.4 for the placement of operators on resources, helps us draw an analogy between the operator placement and the matrix chain ordering problem (MCOP) [89].

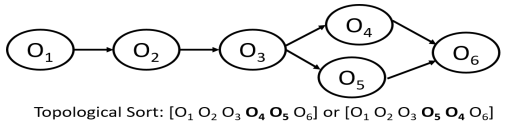

Figure 5.6: Example Operator graphs and their topological order.

This analogy and its significance in reducing complexity of operator placement is explained in the next section 5.3.2.

5.3.2 Solution

We first note that there exists a dynamic programming algorithm for solving MCOP in polynomial time. MCOP is defined as follows: given a sequence of matrices, find the most efficient way to multiply the matrices together. Since matrix multiplication is associative, no matter how we parenthesize the product, the result will be the same. However, the way in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product. Similarly, in our operator placement problem, the way in which we parenthesize operators affects the computation and communications costs. It turns out that the solution of MCOP is also the solution to our distributed operator placement problem.

Instead of enumerating the combinatorial solutions in Figure 5.4, we construct a *placement graph* as shown in Figure 5.5. Each node represents one unique parenthesis in Figure 5.4 and is labeled by one operator or multiple operators running in parallel to complete processing a chunk. An edge from node i to node j in the placement graph corresponds to the ending of one pair of parenthesis and the beginning of the next one in Figure 5.4. It represents the computation cost of operators in node i, the communication cost to transmit output data from node i to node j, and queuing delay at node j. We discuss more details about constructing the placement graph in Section 5.3.3.

The shortest path between the start and the end nodes is the solution of the distributed operator placement problem. For example, if the shortest path is $[(o_1@E \mid\mid o_2@C); (o_3@C)]$, then o_1 is placed on device E while o_2 and o_3 are placed on device C. The intuition behind reduction in the placement problem complexity is that the cost for each parentheses is calculated once even though the cost might be shared between multiple paths. If you look at Figure 5.4, the parentheses $(o_3@C)$ appears in lines 7-10 because it is shared between 4 different paths. However, in the placement graph (Figure 5.5), $(o_3@C)$ is represented as one node and its cost is calculated only once for the 4 paths.

5.3.3 Algorithm Steps

The algorithm can be described in 4 main steps:

Step 1: Topological Sort For a given dataflow graph G, we sort operators in a topological order as shown in Figure 5.6. Topological order has the property that parent nodes appear before their successors, so the order of operator execution is maintained. If a node in the graph has multiple successors, the successors can be ordered arbitrarily. We denote this ordered list of operators as $OpList = \langle o_1, o_2 \ldots, o_n \rangle$. We note that by doing this, we still maintain the original dependency relationship between operations from graph representation in Figure 5.6.

Step 2: Create Placement Graph The list $OpList$ obtained in Step 1 has a sequence of operators that follow certain execution order, which has an analogy with a chain of matrix multiplications. We use parenthesis around adjacent operators in $OpList$ to create unique sublists of size up to the number of containers in all resources. The unique sublists S for $OpList = \langle o_1, o_2, o_3 \rangle$ are:

$$S = \{ \underbrace{(o_1), (o_2), (o_3)}_{\text{parentheses of size 1}}, \underbrace{(o_1\ o_2), (o_2\ o_3)}_{\text{parnethesis of size 2}}, \underbrace{(o_1\ o_2\ o_3)}_{\text{parnethesis of size 3}} \} \tag{5.5}$$

We define a placement set $P(s)$ that denotes the set of possible placements of each subset $s \in S$. For example, $P(s)$ is defined as follows for two examples subsets:

$$P(s = (o_1)) = \{o_1@E, o_1@C\} \tag{5.6}$$

$$P(s = (o_1\ o_2)) = \{(o_1@E \mid\mid o_2@C), (o_1@C \mid\mid o_2@E), (o_1@C \mid\mid o_2@C)\} \tag{5.7}$$

We use the placement sets to generate a placement graph $G_p = (V_p, E_p)$ (See Figure 5.5), where the vertices are the union of all the placement sets: $Vp = \cup_s P(s)$.

Step 3: Add Placement Graph Edges and Edge Weights: Once we have the nodes as shown in Figure 5.5, we start by adding edges from *START* node to all nodes that start with first operator in $OpList$, e.g. o_1. Similarly, we add edges for all nodes that end with last operator in $OpList$ (o_3 in Figure 5.5) to an *END* node. To add edges between intermediate nodes, we look for parent nodes that end with operator i, and we look for the candidate child nodes that starts with operator j, where j is the successor of i in $OpList$. Next, we check the resource that operator j is placed on. If it is placed in a resource that appears in the parent node, we add an edge between the parent node and the candidate child node. The intuition behind adding an edge is that if there is an overlap between compute resources used for the child node and the parent node then the child node has to wait for the resources to be free before it starts execution. Therefore, the weight on the edge denotes the computation, communication. We discuss the details of calculating edge weights in Section 5.3.4

Step 4: Shortest Path: Find the shortest path from *START* to *END* node in the placement graph constructed in Steps 2 and 3. The shortest path gives the full solution of placement and the consumed containers in each resource.

5.3.4 Cost Calculation

In this section, we describe how to calculate the overall computation and communication cost of all operators inside the parentheses such as calculating the cost of $(o_1@E \mid\mid o_2@C)$ in the path $[(o_1@E \mid\mid o_2@C);(o_3@C)]$ from Figure 5.5. We note that due to sequential execution (denoted by ;), the cost of operators inside one pair of parentheses is added to the cost of operators inside the next pair. Hence, in this section we focus on calculating the cost for one pair of parentheses.

Dependent Operators: In this example we show the cost calculation $(o_1@E \mid\mid o_2@C)$. Figure 5.3 illustrates the execution of $(o_1@E \mid\mid o_2@C)$ and the cost is given by:

$$\underbrace{(I-1)e_{1,E}}_{\text{queuing delay}} + \underbrace{e_{1,E}+e_{2,C}}_{\text{execution cost}} + \underbrace{tr(E \xrightarrow{D_{o_1}} C)+tr(C \xrightarrow{D_{o_2}} E)}_{\text{transmission cost}} \tag{5.8}$$

The cost is composed of queuing, execution and transmission costs as illustrated in Figure 5.3, There is an additional term in the transmission cost $tr(C \xrightarrow{D_{o_2}} E)$, which specifies the transmission cost for the output of the current parentheses.

Fork Node: From Figure 5.6, we now consider the cost of $(o_3@E \mid\mid o_4@C \mid\mid o_5@C)$, where the parentheses contain a fork node (i.e., a node with multiple successors) such as o_3. Since the fork node splits the DAG into two or more branches, the cost is given by considering the maximum cost across the different branches so in this example the cost will be the maximum of: (1) Computation and communication costs of o_3 and o_4, and (2) Computation and communication cost of o_3 and o_5.

5.3.5 Algorithm Analysis and Limitations

This analysis represents the scalability aspect of Droplet. It is a summary of a more detailed technical report [87] and the primary contributor to it is Atul Sandur.

Analysis Summary: Among the 4 steps described in section 5.3.3, there are two steps that dominate the complexity of DROPLET: (1) topological sort of input operator graph (Step 1), (2) the shortest path calculation (Step 4) on placement graph. The complexity of topological sort [90] is $O(\mid V_o \mid + \mid E_o \mid)$, where $\mid V_o \mid$ and $\mid E_o \mid$ are the number of operators and links in the operator graph, respectively. The complexity of Dijkstra's algorithm [91] for shortest path calculation when executed on placement graph is: $O(\mid V_p \mid log(\mid V_p \mid)+ \mid E_p \mid)$, such that $\mid V_p \mid$ is number of vertices in the placement graph and $\mid E_p \mid$ is the number of links in the placement graph. For simplicity of analysis, we assume that we have L resources each having one container, hence each vertex in

the placement graph can have up to L operators in the same parentheses. $\mid V_p \mid$ is calculated as the number of sublists of size k multiplied by the number of possible placements of operators in each sublist (See Step 2 in section 5.3.3). In case of L resources and a bracket of size k operators, the number of possible placements is equivalent to the number of ways to obtain an ordered subset of k elements, from a set of L elements (i.e., the number of permutations $[L]k$). Hence, $\mid V_p \mid$ is defined as:

$$\mid V_p \mid = \sum_{k=1}^{L} (\mid V_o \mid -k+1) \cdot [L]k \tag{5.9}$$

As described in [87], $\sum_{k=1}^{L} \mid V_o \mid [L]k$ can be approximated as $\mid V_o \mid \cdot L!$. It is also shown in [87] that $\mid E_p \mid = \mid V_p \mid \cdot L!$. Hence, the overall complexity of DROPLET can be given by:

$$O(\mid V_o \mid \cdot L! \cdot (log(\mid V_o \mid L!) + L!) + \mid E_o \mid) \tag{5.10}$$

Limitation of DROPLET: Going back to Figure 5.5, our approach has an implicit assumption that when o_1 and o_2 are executed concurrently in $(o_1@E \mid\mid o_2@C)$, they will fully utilize available resources and that's why o_3 in $(o_3@C)$ has to wait for both operators to finish execution before starting execution on device C. An improved model would account for the fact that o_3 can start right after the fastest of o_1 and o_2 and provide better resource utilization. We plan to improve this model as part of future work.

5.4 EVALUATION

In this section, we implement DROPLET algorithm and evaluate its performance against the closest algorithms in the literature using various applications from different domains.

Compared approaches. We compare four methods for operator placement:

1. **DROPLET (Our Approach):** We use our algorithm that we described in 5.3.3 to decide the operator placement.

2. **Brute Force (BF)**: We execute all different placements and we empirically measure the completion time of all the operators then we report the placement with the smallest execution time.

3. **SCPP (Single-User computation partitioning) [68]**: We implement the SCPP approach which finds an optimal solution for the placement problem with some assumptions such as (1) Operator graph is a sequence of operations (i.e., not a DAG or a tree), (2) Pipeline parallelism is not considered (i.e., all the video frames in one chunk have to be processed by

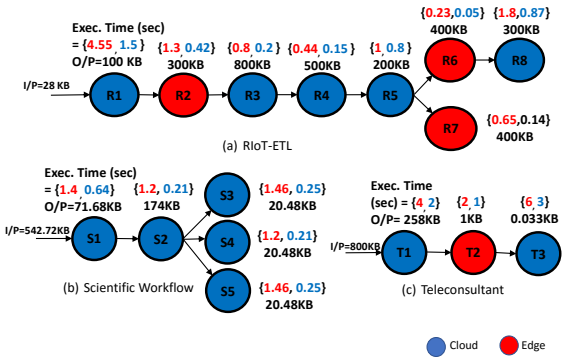

Figure 5.7: Operator graphs for real world applications. Color coding of operators shows DROPLET's placement output

operator 1 before they move together to operator 2). We use this approach to evaluate the significance of pipeline parallelism.

4. **Genetic algorithm (GA)**: We implement genetic algorithm as a representative of search strategies that have been proposed in the literature [69] [85] to solve the placement problem. DROPLET is compared with GA when the number of operators is significantly high (i.e., more than 100 operators) to evaluate it's scalability and performance characteristics at large scale. We use the fitness function that we modeled in Equation 5.3 and we use similar parameters to the ones reported in [69]: population size=50, crossover probability=0.5, mutation probability=0.15. However, for comparison purposes we terminate GA after it runs for 10 times the duration taken by DROPLET to obtain shortest path placement.

Applications. We evaluate DROPLET through three real-world applications from various domains:

1. Teleconsultant [92]: a Telehealth application in which paramedics wear body cameras to analyze video feed of patients to detect strokes. Analysis is done using an edge device in the ambulance and a cloud server in the hospital. The application consists of three operators, and we use a stream of images obtained from the image gallery for facial paralysis [93] with chunks of 10 images.

2. RIoT-ETL [39]: an IoT benchmark that performs data preprocessing and cleaning on the incoming data stream. The benchmark is modeled as a dataflow graph of 8 operators as shown in Figure 5.7. In our experiments, we generate data that mimics the original dataset and use a chunk size of 10 observations.

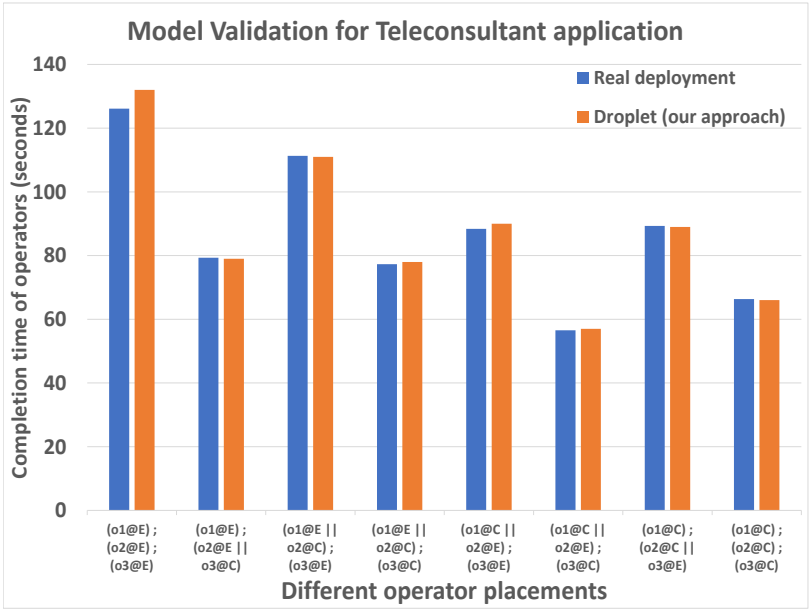

Figure 5.8: Comparing model estimated completion time for *Teleconsultant* application with observed times during deployment of different operator placements

3. Scientific Dataflow [94]: We use a sub-dataflow of the *LIGO* scientific workflow that processes astronomy data. [94] provides the profiling measurements for computation and communication cost for the 5 operators that comprise the sub-dataflow (as shown in Figure 5.7). We generate data that follows same profiles and we use a chunk size of 10 observations.

System setup: The computing infrastructure consists of edge and cloud resources. We use 1 laptop as edge device and one server as the cloud. The edge device has Intel Core i7-5600 CPU with 4 GB of memory and the cloud server has Intel Core i7-3770 CPU with 8 GB of memory. We use one thread on the edge device and one thread in the cloud server and control the bandwidth from edge to cloud server to be 12 Mbps and from cloud server to edge device to be 13 Mbps, which simulates an average wide area network connection. Each of the edge and cloud servers has a local dataflow engine, Apache NiFi, that handles execution of operators that are deployed on it. Nifi is an engine designed for composing user-defined operators and executing dataflows in a single machine or across multiple machines. Each of the edge and cloud servers has a local deployment of Nifi and we use Echo [39] orchestration framework to handle the communication between the two Nifi instances. Echo does the following: (1) receives the operator graph from the user, (2) reads the resource graph from a directory of registered resources, (3) communicates

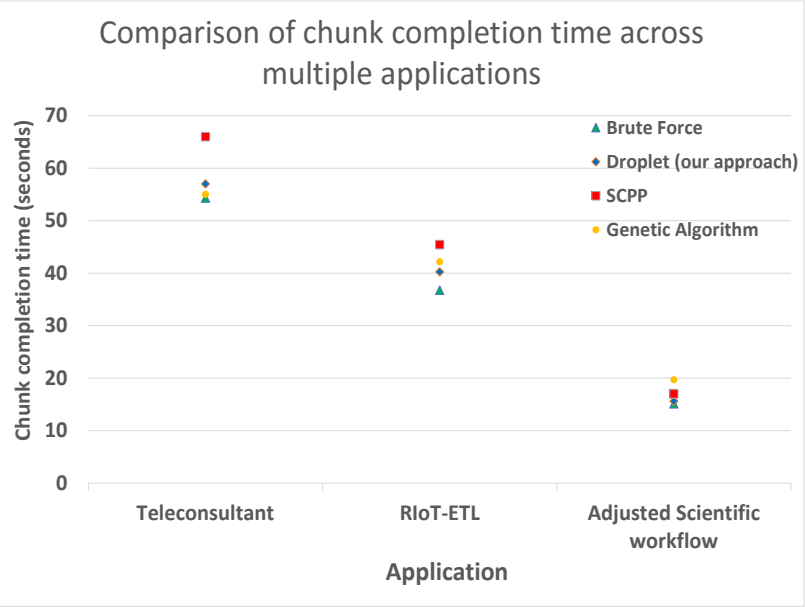

Figure 5.9: DROPLET is evaluated against other approaches for performance in finding the operator placement with shortest chunk completion time, across multiple applications

with DROPLET to find out the mapping from operators to resources, (4) deploys the operators that belong to the edge and cloud resources in their local Nifi engines, and (5) manages communication of intermediate results between Nifi engines.

Evaluation Metrics: We conduct experiments to evaluate DROPLET performance in terms of:

- *Model Accuracy:* For each placement, we compare our estimate of completion time of operators with observed completion time in the real system deployment and evaluate accuracy of our estimates.

- *Chunk Completion time*: Given an operator graph and a resource graph, we deploy the operators according to the placement obtained from DROPLET that provides the shortest completion time of a chunk of data frames and measure it. This is compared against chunk completion time from other approaches.

- *Time to find Placement*: We measure the time that DROPLET takes to obtain the placement decision and focus on its behavior for large scale operator graphs.

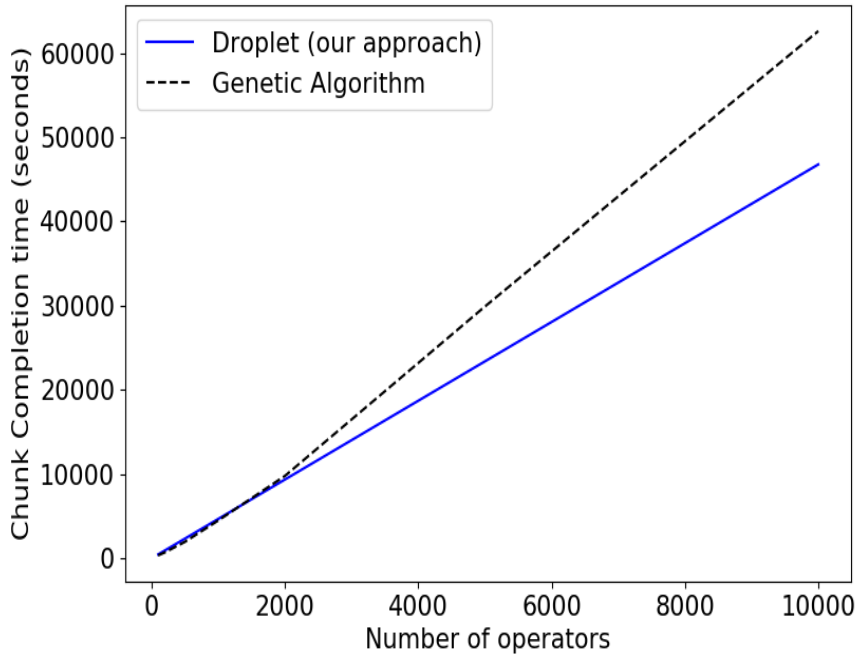

Figure 5.10: Comparison with genetic algorithm with increasing number of operators

5.4.1 Model Accuracy

In order to show that our modeling of completion time for operator graph execution closely reflects the observed running time from real-world deployments, we compare the two measurements for our applications. Figure 5.8 shows the results for the *Teleconsultant* application. As we can see, the estimated completion times for each of the possible operator placements in the application are very close to empirically measured values. The average error in estimated completion time across placements is only about 0.42% and Pearson correlation coefficient is 0.998.

5.4.2 Chunk Completion Time

Real-World Applications: To determine how well DROPLET does on real-world applications, we solve the operator placement problem with different approaches and we plot the results in Figure 5.9. BF gives us the reference minimum time for each application because it is obtained from an exhaustive search over all placements. So it was feasible to use BF to obtain optimal completion time in order to quantify the error introduced by our modeling approximations. SCPP provides the execution time when the operators in the input graph are executed in a sequence with no pipeline parallelism across inputs. The results show that DROPLET is within an average of

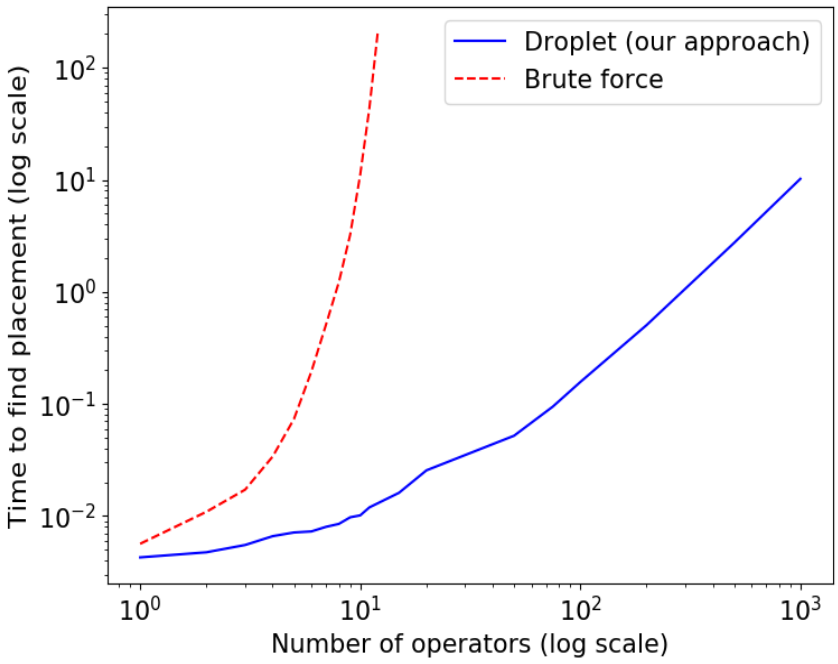

Figure 5.11: Scalability in time to search for the best placement with increasing number of operators

4.09% error compared to Brute Force, in finding the operator placement with minimum execution time. Moreover, DROPLET and GA are able to leverage pipeline parallelism to find placements that outperforms SCPP in two of the above applications. We note that DROPLET and SCPP give the same execution time for the Scientific dataflow. The reason is that the ideal placement for this application is to place all the operators in the cloud. This reduces the significance of the pipeline parallelism that happens when some operators are running in the edge and other operators are concurrently running in the cloud.

Large Scale of Operators: In case of large scale of operators, it is infeasible to find the optimal placement solution via BF. So we compare the placements, obtained by DROPLET, with GA, by increasing the number of operators. Figure 5.10 shows a comparison between the chunk completion for both algorithms. The results show that as the number of operators increases, the search space that GA has to navigate, becomes huge (in the order of 2^n) and GA finds less optimal results. This result confirms that as the number of operators becomes of the order of hundreds which is the case in several Deep Neural Networks models [95], DROPLET is able to find placements that are more than 20% better than GA in terms of completion time while being 10 times faster.

5.4.3 Time to find Placement

We now analyze the scalability characteristics of DROPLET with increasing numbers of operators. Figure 5.11 compares DROPLET with BF, in the time to find shortest path placement of operators. This time includes the construction time of placement graph and executing shortest path algorithm. With BF, we are not able to complete the search in a reasonable amount of time when number of operators increased beyond 15. DROPLET however is able to complete the search within 10 seconds for 1000 operators. It is shown to scale polynomially with increasing the number of operators.

5.5 CONCLUSION

In this chapter, we studied the problem of distributed operator placement for IoT applications. We studied the scenario where IoT applications are sharing geo-distributed resources across the edge and cloud. We addressed several challenges that are unique to IoT applications such as: (1) processing continuous stream of inputs in pipelining fashion, and modeling the queuing delay involved in this scenario. (2) Modeling the trade-off between computation and communication delays when placing operators in heterogeneous and geo-distributed compute resources We formulated our problem as a shortest path problem that navigates various possibilities of operator placement and chooses the one with the minimum total cost with respect to operator graph completion time. We showed through analysis and experimentation that our solution scales log-linearly in the number of operators.

CHAPTER 6: COSTLESS: EDGE-CLOUD PRICE OPTIMIZATION ALGORITHM FOR SERVERLESS COMPUTING PLATFORMS

In chapter 5, we addressed the problem of deploying visual IoT applications among edge and cloud compute resources that are privately owned. In this chapter, we extend the work to support deploying the applications in public cloud infrastructure where the price becomes an important aspect in the decision of how to deploy the application. In this work, we study the newest generation of public cloud providers offerings known as Serverless computing [17]. Serverless computing is becoming an increasingly popular platform for IoT analytics due to the low provisioning overhead, however, it comes with new pricing model that we study in this chapter.

The pricing model of serverless computing depends on the memory allocated to the functions and the CPU time of executing them. In addition, serverless computing services, such as AWS lambda, provide a method to create a *workflow* of functions, called *state machine*. The state machine specifies the order at which lambda functions are invoked such that the output of one function is the input of the next function. AWS lambda charges an additional price for each transition from one function to another. Therefore, one way to optimize the price is to *fuse* multiple functions together and rewrite them as one function to avoid paying for the transition price. However, it is not always ideal to fuse functions when they have different memory requirements. For example, if one function requires 2GB of memory and takes one second to execute and the next one requires 0.5 GB but needs 5 seconds to execute, fusing them requires executing one long function (6 seconds) with at least 2GB of memory which is not price-effective as we will describe in Section 6.1. Therefore, the decision of which functions to fuse is non-trivial problem especially in the presence of large workflows with more than 10 functions such as scientific workflows [94] and machine learning models [95]. We refer to the problem of deciding which functions to fuse as the *Function Fusion Problem*.

Another challenge in serverless computing is the *Function Placement*. In order to match the increasing volume of data coming from Internet of Things (IoT) devices, AWS offers another service in its serverless computing ecosystem, called AWS Greengrass [19]. AWS Greengrass allows processing data closer to the source where the data is generated instead of sending it across long routes to data centers or clouds. Greengrass supports running functions on edge devices (e.g., Raspberry Pi) that are controlled by the users and provide a tight integration between the user's edge device and the cloud infrastructure owned by Amazon. Users of Greengrass are charged per device rather than per function so no matter how many functions are running on the edge device, the price is fixed. However, due to the limited compute capacity on such edge devices, the function execution might be significantly slower. A natural question that arises is which functions to place on the resource constrained edge devices in order to optimize the price without dramatically

increasing the latency. We refer to this problem as the *Function Placement* problem.

In this chapter we address the problems of *Function Fusion* and *Function Placement* in order to optimize the price of deploying serverless applications in AWS lambda. We start by highlighting the different factors that affect the price of AWS Lambda as a representative serverless computing service We then formulate the problem of optimizing the price and execution time of serverless applications. We propose two models: (1) price model for AWS Lambda, and (2) execution time model that estimates the response time of the workflow of functions based on their execution and communication costs. Finally, we present an algorithm to explore possible function fusions and placements. We represent the solutions in a structure that we refer to as the *Cost Graph* and we formulate the problem as a Constrained Shortest Path problem in which we find the solution with the best latency within a certain budget and vice versa.

The rest of the chapter is structured as follows. In Section 6.1, we start with the background about serverless computing pricing and we highlight the factors affecting it. In Section 6.2, we present an overview of the optimization goals and the tradeoffs that we address in this chapter. In Section 6.3, we formally define the pricing model, execution time model, and the cost optimization problem. In Section 6.4, we present the novel function-fusion placement algorithm. In Section 6.5, we evaluate our algorithm and we compare it with the optimal solutions and other heuristics. Finally, Section 6.6 concludes the chapter.

6.1 BACKGROUND AND MOTIVATION

6.1.1 AWS Lambda Pricing

Figure 6.1 shows an image processing state machine (workflow) with five functions. Each node in the workflow is an AWS Lambda function and the arrow describes the dependency between functions. The workflow starts by detecting the face in the photo and then matches the face against a collection of previously indexed faces. The photo is then resized to be shown as a thumbnail in the smartphone application, the user's face is indexed in a collection for future matching, and finally the photo's metadata is saved in the user profile.

The price for each lambda function is calculated using 4 factors:

1. The number of times each function is executed per month (e.g., 1,000,000 executions/-month).

2. The memory allocated to the function by application developers. The CPU resources allocated to the function represent an implicit parameter. This parameter value is proportional

to the function's allocated memory (i.e., a 256 MB function is automatically allocated twice the CPU speed than a 128 MB function).

3. The duration how long the function runs (e.g., 2 seconds).

4. The price per 1 GB of memory and 1 second of execution. For AWS Lambda, the price of 1 GB and 1 seconds is 0.00001667$/GB-s.

Figure 6.1 shows the memory allocation and duration for each function. Assuming that the workflow is executed 1,000,000 times, the price of the first function *FaceDetection* is calculated as:

$$\begin{aligned} Price_{FaceDetection} = 1,000,000\ execution\ *\ 512/1024\ GB \\ *\ 2\ seconds * \ 0.00001667\$/GB-s = 16.67\$ \end{aligned} \tag{6.1}$$

Similarly, the price for the five functions is given as :

$$\begin{aligned} Price_{lambdaFunctions} = 1,000,000 * 0.00001667[(512/1024\ *\ 2) \\ +\ (128/1024\ *\ 5)\ +\ (128/1024\ *\ 1.5)\ + \\ (256/1024\ *\ 0.3)\ +\ (128/1024\ *\ 0.2)] = 35\$ \end{aligned} \tag{6.2}$$

In addition to the lambda functions price, there is an additional price for each transition between functions, referred to as a state transition price. The state transition price is charged by AWS to handle the message passing and coordination between two successive functions.

The workflow in Figure 6.1 has 6 state transitions defined by the number of arrows. The total states transitions price is:

$$\begin{aligned} Price_{transition} = 6\ transitions\ per\ execution\ *\ 1,000,000 executions \\ *\ 0.000025\$\ state\ transition\ price = 150\$ \end{aligned} \tag{6.3}$$

The total price of the entire workflow includes both the lambda function price (Eq. 6.2) and the state transition price (Eq. 6.3) and it is given by:

$$Price_{workflow} = Price_{lambdaFunctions} + Price_{transition} = 35 + 150 = 185\$ \tag{6.4}$$

As we mentioned earlier, AWS Greengrass is another service that charges a small per-device fee

to connect user-owned edge devices securely to Amazon cloud and no matter how many functions are executed on the edge device, the price does not change. The price of one edge device ranges from 0.16$ to 0.22$, based on the region.

6.1.2 Factors Affecting Price of Serverless Applications

Based on the pricing model described above, we identify three major factors that are crucial to the pricing of serverless application workflows. Such factors are : (1) Number of State Transitions, (2) Edge vs. Cloud Computation, (3) Memory allocated to each cloud function.

In the following, we describe each factor in detail and in the rest of the chapter we focus on manipulating the first two factors to optimize the price of serverless applications.

Number of State Transitions: Building applications from individual components, where each component performs a small function, makes applications easier to scale and change. However, we note that the transition price can sometimes dominate the price of the entire workflow as shown in Eq. 6.6. In such cases, there are incentives to reduce the number of state transitions to make the price lower and within a certain budget without affecting the application correctness.

An effective method to reduce the state transitions is to fuse multiple functions to form one bigger function. For example, in Figure 6.1, the first two functions, *FaceDetection* and *Check-FaceDuplicate*, can be fused together to be one function *FaceDetAndDup* and remove the state transition between them, which could potentially reduce the cost by 25$ ($1\ statetransition\ *\ 1M\ executions\ *\ 0.000025\$$). However, it is not necessarily useful to fuse functions since one function requires 512MB and the other function requires 128MB, and fusing them together will require using at least 512MB for the fused function. Hence, assuming that the second function will still run for a duration of 5 seconds, the cost of the fused functions will then be:

$$P_{FaceDetAndDup} = 1,000,000\ *\ 512/1024\ *\ (5+2)\ seconds\ *\ 0.00001667\$ = 58.3\$ \tag{6.5}$$

and the previous cost of the non-fused functions used to be :

$$\begin{aligned} P_{FaceDet} + P_{FaceDup} + P_{transition} = 1,000,000 * 0.00001667[(512/1024\ *\ 2\ seconds) \\ + (128/1024\ *\ 5)] + 25\$ = 52.3\$ \end{aligned} \tag{6.6}$$

Hence, in this case the fused function will end up being more costly than original functions but in other cases when both functions have the same memory requirements fusing them can reduce the overall cost .

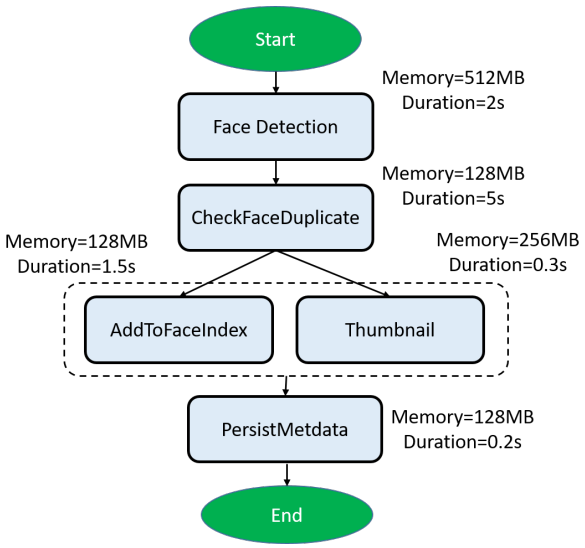

Figure 6.1: Example AWS workflow (state machine)

Another challenge for the function fusion operation is when trying to fuse parallel functions with their parent. For example, Figure 6.1 has two parallel functions *AddToFaceIndex* and *Thumbnail*. If one or both of them are fused with their parent, then fusion will cause two parallel functions to run sequentially and the latency of the entire workflow increases.

We conclude that it is not trivial to decide which functions to fuse because it could have implications on the price and the latency. In our model we consider both the price and latency of the fused functions and we decide to fuse functions that can keep the cost under a certain budget while maintaining the best possible latency within the budget constraints.

Edge vs. Cloud Computation: Computing functions on edge devices could be cost effective because no matter how many functions you execute on it, the charge is per-device only and it is relatively cheap ($0.16\$ - 0.22\$$ per device per month). The edge device typically communicates with the cloud through saving the intermediate data in Amazon's Simple-Cloud-Storage-Service (Amazon S3). There is an additional price for storing the data on S3 but it is also relatively cheap ($0.023\$$ per GB per month). Due to the limited compute capacity of such edge devices, the function execution might be considerably slower. Therefore, if the application is compute-intensive, it is desirable to place only a subset of the functions on the edge to keep the latency within certain bounds.

We further note that there is a non-trivial transmission time to send the intermediate data from the edge to the cloud. Therefore, it is desirable to place the functions that reduce the transmission time on the edge device. In our model we consider both computation and transmission times and we choose the best placements with respect to both price and latency.

Memory Allocation for each Function: AWS Lambda allows developers to allocate memory

for their function. The CPU resources are not directly configurable because AWS allocates CPU proportional to the allocated memory. For example, AWS allocates twice as much CPU power for a function while going from 128MB to 256MB of memory. However, based on the function implementation, and whether it is compute-intensive or not, if it runs for 4 seconds with 128MB, it may not run for exactly 2 seconds when it is switched to 256 MB. Some functions may run faster than 2 seconds and some may run somewhere between 2 and 4 seconds and after increasing the memory to a certain value, the execution time tends to stabilize because the code does not fully utilize the CPU.

Tuning the memory for each function separately is a difficult problem and directly impacts the price and the latency of serverless applications. We note that exploring different memory configurations is fundamentally similar to exploring different placements of the function between edge and cloud resources. Intuitively, placing a function on an edge device is fundamentally similar to placing a function on a VM with 128MB or a VM with 512MB. The only effect is that the execution time changes. Hence, in our algorithm we not only explore placing functions on edge devices but we can also explore placing them on 128MB cloud VM or 256MB cloud VM. For the simplicity of discussion, we focus for the rest of the chapter on one edge and one cloud configuration.

6.2 OPTIMIZATION GOALS

In this section we present an overview of: (1) The utility that we aim to optimize, (2) the applications that we addres, (3) the underlying compute resources that we levarage, and (4) the tradeoffs that we explore:

Utility: We optimize the billing price for processing a stream of data by an analytics application.

Application: A sequence/graph of AWS Lambda functions.

Compute resources: (1) Edge devices owned-by the user and managed by the public cloud provider, (2) remote cloud resources owned and managed by the public provider.

Tradeoffs: We aim to control the tradeoff between the application's price, the application's end-to-end latency, and the application's locality (i.e., the ability to process the entire application closer to where it is generated)

6.3 MODELS AND PROBLEM DEFINITION

We start by describing the resource model, data model, and the workflow model. Then we formally define the problem. In Section 6.4, we present the intuition behind our approach and the algorithm to solve it. In both sections, we follow the notation in Table 6.1.

```
{
  "userId": "user_1",
  "s3Bucket": "face-collection-bucket",
  "s3Key": "face_photo.jpg"
}
```

Figure 6.2: Example input request to Lambda function

6.3.1 Resource Model

We consider two components of the serverless computing platforms one edge E and one cloud C. The edge resource is a device close to the data sources (i.e., IoT devices) and owned by the user. An example of such edge devices are Raspberry Pi, and personal desktop. The edge devices use Greengrass core software that provides a tight integration between the edge device and AWS cloud infrastructure. The price of connecting the edge device to AWS Lambda is p_E dollars per month. The cloud resources are following the AWS Lambda resource model in which the user specifies a set of functions $\{f_i \mid i = 1 \dots n\}$. For each function the user requests a memory $m_{i,C}$ that will be allocated to the container/VM executing the function f_i. The user does not need to worry about the VM executing the function and how they are provisioned, the pricing only depends on the memory and the duration as we will describe in the pricing model (Section 6.3.5). We note that there are other resources allocated to the function such as timeout, and maximum concurrency. However, in this work we only focus on the memory resource and we assume that the function runs in finite duration and within a maximum concurrency equivalent to the default concurrency (max concurrency = 1000) which means that the function cannot serve more than 1000 requests at a time.

6.3.2 Data Model

The data sent to the Lambda function is in the form of a request encoded in JavaScript Object Notation (JSON) format. Since JSON is a text-based format, it can directly encode text data such as text files or sensor readings. However, if the data is in binary format such as a compressed image, then the image is first uploaded to a persistent storage (e.g., Amazon S3) and the JSON request will encode the location of the uploaded image as shown in the example JSON request in Figure 6.2. The number of requests per month is denoted by r. We do not make any explicit assumptions on whether the requests come as a continuous stream or they come in bursts.

n	=	Total number of functions
r	=	Total number of executions of a workflow
G_f	=	Input function graph
G'_f	=	Fused function graph
f_i	=	Function i in graph G_f
f'_i	=	Fused function i in graph G'_f
X_i	=	Placement variable (1: f_i on cloud, 0: f_i on edge)
t_i	=	Completion time of function i
$e_{i,C}$	=	Execution time of function i on the cloud
$e_{i,E}$	=	Execution time of function i on the edge
D_{f_i}	=	Size (bytes) of output data of function f_i
$B_{E,C}$	=	Bandwidth (bytes/sec) between edge and cloud
$tr(E \xrightarrow{D_{f_i}} C)$	=	Transmission time (sec) between edge and cloud
$s_{i,C}$	=	Time to schedule function i on the cloud
$m_{i,C}$	=	Memory allocated to function i
m_i	=	Maximum memory used by function i
p_E	=	Price of connecting one edge device to AWS cloud
p_s	=	Price of one state transition
$p_{m_{i,C}}$	=	Price of 1 sec exec. of function i with memory $m_{i,C}$
$P(G_f, X_{i=1,..,n})$	=	Price of workflow G_f according to $X_{i=1,..,n}$
$T(G_f, X_{i=1,..,n})$	=	Execution time of G_f according to $X_{i=1,..,n}$

Table 6.1: Table of Notation

6.3.3 Workflow Model

We model the workflow in AWS Lambda as a directed-acyclic-graph (DAG) $G_f = (V_f, E_f)$ of functions (Fig. 6.3). A function is a processing element that can execute user-defined code (e.g. convolution, face detection). The vertices in the DAG represent functions $V_f = \{f_i \mid i = 1 \ldots n\}$, and the links between them $E_f = \{f_i \rightarrow f_j \mid i \neq j\ ,\ 1 \leq i, j \leq n\}$ represent the workflow dependencies, where $f_i \rightarrow f_j$ means f_i is executed before f_j and the output of f_i is the input of f_j.

6.3.4 Function Profile

Each function in Figure 6.1 is associated with a *profile* which includes:

1. The cost of executing function f_i when placed on node E or C. Function f_i can be placed on node E or C so we denote their corresponding execution costs as $e_{i,E}$ and $e_{i,C}$. We assume that each request has equal sized data (e.g., 720p images), therefore the execution cost is the same for all requests.

2. The size of output data of function f_i is denoted as D_{f_i} bytes. The transmission time of the

result of f_i from the edge to the cloud is $tr(E \xrightarrow{D_{f_i}} C) = \frac{D_{f_i}}{B_{E,C}}$, where $B_{E,C}$ is the bandwidth (bytes/sec) from the edge to the cloud.

3. The maximum memory consumed by function f_i, is denoted as m_i. We note that AWS Lambda has only a discrete set of memory values that can be allocated to a function (e.g., 128 MB, 256MB, 320MB, 384MB,...). If the actual memory consumption m_i, reported by AWS, is not equivalent to any of the allowed values, a user has to set allocated memory $m_{i,C}$ to the closest allowed value that is larger than m_i. Therefore, if $m_i = 340MB$, we assume that the allocated memory is $m_{i,C} = 384MB$.

4. The scheduling delay $s_{i,C}$ of the function. When AWS receives requests to run a function, it reports the timestamps about the following timed events: (1) Request received, (2) Function scheduled for execution, and (3) Function started execution. The scheduling delay is the time between receiving the request and starting to execute the function.

6.3.5 Price Model

For each function f_i, we define the variable X_i which takes a binary value. X_i equals to 1 when f_i is executed on the cloud and it takes a 0 value when f_i is executed on the edge device. We note that the edge device has a fixed cost p_E for connecting it to the AWS cloud no matter how many functions are allocated to it. On the other hand, the price of executing f_i on the cloud depends on the memory $m_{i,C}$ allocated to it, and its execution time $e_{i,C}$. The price per 1 GB memory and 1 sec of execution time is denoted as $p_{m_{i,C}}$. The price for each state transition (i.e., each link in Figure 6.1) is denoted as p_s. We formulate the price per month P as follows:

$$P(G_f, X_{i=1,..,n}) = \sum_{i=1}^{i=n} X_i \cdot r \cdot e_{i,C} \cdot m_{i,C} \cdot p_{m_{i,C}} + r \cdot (n+1) \cdot p_s + p_E \tag{6.7}$$

We note than the number of transitions for n functions is $n+1$ by calculating the start and end transitions as shown in Figure 6.1, entire workflow of functions is executed for r times then the number of transitions is also multiplied by r, so the total number number of executed transitions is $r(n+1)$. We also note that AWS offer some requests and transitions free of charge in the beginning of each month but our pricing model only considers the price after the free requests are consumed.

6.3.6 Execution Time Model

In this section, we formulate the execution time for each request. The execution time for a request is defined by the completion time of the last function f_n minus the starting time of the first

function f_1. We denote the completion time of f_n as t_n. The total execution time T of the workflow is given by:

$$T(G_f, X_{i=1,..,n}) = t_n(G_f, X_{i=1,..,n}) - t_0 \tag{6.8}$$

such that t_0 is the time before starting the execution of f_1. The completion time of function f_i is given by the recursive formula:

$$\begin{aligned} t_i(G_f, X_{i=1,..,n})) = & \underbrace{t_{i-1}(G_f, X_{i=1,..,n})}_{\text{completion time of prev. function}} + \underbrace{(1-X_i)\cdot e_{i,E}}_{\text{Execution time on edge}} \\ & + \underbrace{\mid X_i - X_{i-1} \mid \cdot tr(E \xrightarrow{D_{f_{i-1}}} C)}_{\text{Transmission time}} + \underbrace{X_i \cdot (e_{i,C} + s_{i,C})}_{\text{Execution time on cloud}} \end{aligned} \tag{6.9}$$

Such that $tr(E \xrightarrow{D_{f_{i-1}}} C)$ is the transmission time of the intermediate data from f_{i-1} to f_i. We assume that the transmission time between two functions running on the cloud or two functions running on the edge is negligible.

6.3.7 Problem Definition

Let $G'_f = (V'_f, E'_f)$ be the new function graph after function fusion, such that the vertices represent fused functions $V'_f = \{f'_i \mid i = 1 \ldots m\}$ and each fused function is a concatenation of two or more functions $f'_i = f_1 \mid f_2 \mid f3 \mid ..$, where the symbol "|" denotes concatenation. Let X'_i be the placement variable $\{X'_i \mid i = 1 \ldots m\}$ for each fused function f'_i. Given the function graph G_f, we define the cost optimization problem as finding the fused graph G'_f and the placement variables $\{X'_i \mid i = 1 \ldots m\}$ such that the price $P_{G'_f, X_{i'=1,..,m}}$ is minimized and the execution time does not exceed a certain threshold T_{thresh} so the problem can be formulated as:

$$\text{minimize } P_{G'_f, X_{i'=1,..,m}} \quad \textit{where} \quad T_{G'_f, X_{i'=1,..,m}} < T_{thresh} \tag{6.10}$$

6.4 APPROACH

One of the key contributions in our approach is to jointly represent the solutions for the function placement and function fusion in one graph which we refer to as the *Cost graph*. In order to illustrate how we build the *Cost graph*, we take an example function graph G_f with three functions $V_f = f_1, f_2, f_3$ as shown in Figure 6.3 and two different resources: one edge device E and one

Figure 6.3: Example Workflow with three functions

1. (f1 @ C)(f2 @ C)(f3 @ C)
2. (f1 @ C)(f2 @ C | f3@C)
3. (f1 @ C | f2 @ C)(f3@C)
4. (f1 @ C | f2 @ C | f3 @ C)
5. (f1 @ E)(f2 @ C)(f3 @ C)
6. (f1 @ E)(f2 @ C | f3 @ C)
7. (f1 @ E | f2 @ E) (f3 @ C)
8. (f1 @ E | f2 @ E | f3 @ E)

Figure 6.4: Feasible function placement and function fusion solutions. Each line is one solution. Symbol "|" denotes fusion

cloud node C. We assume that intermediate output data can flow from E to C but not vice versa.

Feasible Solutions: Each solution requires deciding which functions to fuse if any (Function fusion) and assigning each fused function to E or C (Function placement). We show the possibilities of function fusion and placement in Figure 6.4. Each row is a possible solution to the cost optimization problem. The label $f_1@E$ means that f_1 is placed on device E, similarily the label $f_1@C$ means that f_1 is placed on the cloud. The functions inside parentheses show the functions that are being fused and the symbol "|" denotes fusion operation.

Line 1 in Figure 6.4 shows the possibility that the three functions remain unchanged and placed on the cloud. Lines 2-4 show different possibilities of fusing functions while remaining on the cloud. Lines 5-7 show some possibilities for partitioning the functions across E and C. Line 8 shows an extreme case when all functions are fused together and placed on the edge. We note that we prune some of the solutions such as $(f_1@E)(f_2@E)(f_3@E)$ because it is equivalent to line 8 (i.e., fusing the three functions and placing them on the edge). We note that placing three functions on E without fusing them neither have a price or execution time benefit. We also prune solutions like $(f_1@C)(f_2@E \mid f_3@E)$ because the data flows from E to C but not vice versa.

Cost Graph Representation: Instead of enumerating the possible solutions in Figure 6.4, we construct a *cost graph* as shown in Figure 6.5. Each node represents one unique parenthesis from Figure 6.4 and is labeled by a fused function. A link from node u to node v in the cost graph corresponds to the ending of one pair of parenthesis and the beginning of the next one in Figure 6.4. A link in the cost graph represent the transition from one fused function to another and it holds two independent costs:

1. **Price Cost** c_{uv}: this includes the price of fused function i and the state transition from i to j.

2. **Delay Cost** d_{uv}: the execution time of fused function i and the transmission time of output data from i to j.

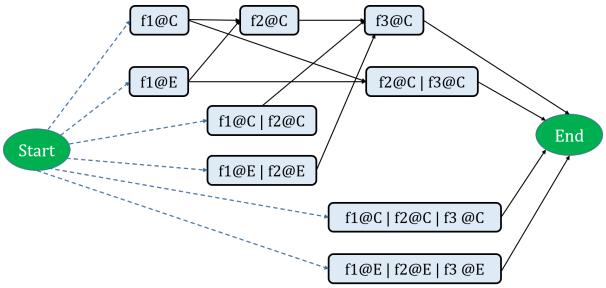

Figure 6.5: Cost Graph, each path represent a solution for function placement and fusion. Each edge include execution time and price costs

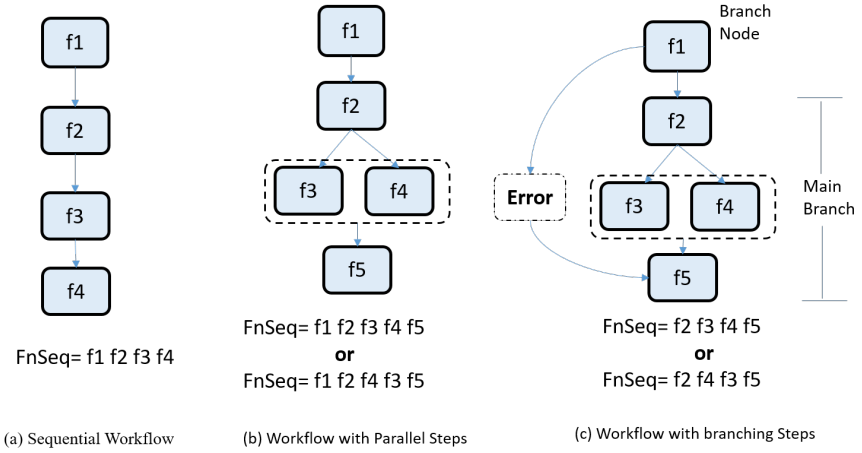

Figure 6.6: Different workflow types supported by AWS

We will discuss more details about constructing the placement graph and calculating the link weights in Sections 6.4.1 and 6.4.2 .

Intuitively, the shortest path between the start and end nodes is the solution of the cost optimization problem. For example, if the shortest path is $[(f_1@E)(f_2@C \mid f_3@C)]$, then f_1 is placed on device E while f_2 and f_3 are fused into one function and placed on the cloud. The challenge to solve this problem is that each link has two independent costs (i.e., price and time) which makes it infeasible to solve the problem using the standard shortest path algorithms such as Dijkstra [91]. Therefore, we formulate the problem as a constrained shortest path problem (CSP).

Problem Transformation (Constrained Shortest Path):

Let us consider a cost graph $G_c(V_c, E_c)$, each link $(u,v) \in E_c$ is associated with two costs: a price cost c_{uv}, and a delay cost d_{uv}. Let s and t be two distinguished nodes in the graph. For each path, let us denote the set of all paths from s to t as Y_{st}. For any path $y \in Y_{st}$, we define:

$$c(y) = \sum_{(u,v)\in y} c_{uv} \; and \; d(y) = \sum_{(u,v)\in y} d_{uv} \tag{6.11}$$

Given $T_{thresh} > 0$, let $\overline{Y}_{st}$ be the set of all s-t paths y such that $d(y) \leq T_{thresh}$. The CSP problem is to find a path $y^* = argmin\{c(y) \mid y \in \overline{Y}_{st}\}$. In other words the problem is to find the path with the minimum price such that the delay does not exceed a threshold T_{thresh}.

6.4.1 Costless Algorithm Steps

In this section, we described the steps of Costless which include handling different workflow types, constructing the cost graph, and solving the constrained shortest path problem.

Step 1: Create an intermediate representation from different workflow types: AWS Lambda provides an API to define the the application workflow as a series of steps. Figure 6.6 shows examples of the flow of steps which include sequential, parallel, and branching steps. We try to represent the workflow as a sequence of functions $FnSeq$. Figure 6.6(a) shows the simplest case which is the sequential steps. We can represent the $FnSeq$ based on the order of functions from *Start* to *End*. Figure 6.6(b) shows a workflow with parallel steps in which multiple functions are executed in parallel and their outputs are aggregated before the next function starts. Similar to the sequential workflow, we represent the $FnSeq$ based on the order of functions from start to end and we arbitrarily order the parallel steps. We note that in addition to the $FnSeq$, we keep track of the original DAG representation while calculating the cost in the cost graph to take the maximum execution time as the reference execution time for the entire parallel step, and we consider the sum of the prices of parallel function as the price for the entire parallel step.

The last type of workflows that we deal is the workflow with branching steps shown in Figure 6.6(c). Such workflows typically have a branch node that has a condition to decide in which branch the execution will proceed. Since, we are interested in optimizing the cost for the general case, we only focus on the main branch that contains the functions rather than the branch that contains the error handling. Therefore, we represent the $FnSeq$ based on the main branch only. In Figure 6.6(c), the main branch is similar to the workflow with parallel steps in Figure 6.6(b) and the $FnSeq$ representation follows the same method.

Step 2: Construct cost graph: The list $FnSeq$ obtained in Step 1 has a sequence of functions that follow certain execution order. We use parenthesis around adjacent functions in $FnSeq$ to create fused functions of size up to the length of $FnSeq$ For example, the set of possible fused functions F' for $FnSeq = \langle f_1, f_2, f_3 \rangle$ are:

$$F' = \{ \underbrace{(f_1),(f_2),(f_3)}_{\text{orginal non-fused functions}}, \underbrace{(f_1\ f_2),(f_2\ f_3)}_{\text{fusing two functions}}, \underbrace{(f_1\ f_2\ f_3)}_{\text{fusing three functions}} \} \tag{6.12}$$

We define a function $L(f')$ that denotes the set of possible placements of function f'. In general,

each function f' that starts with f_1 can be placed on either edge device E or cloud C. However, if f' does not start with f_1 then it can only be placed on C because the data does not flow from C to E. In the following we show some examples of $L(f')$

$$L(f' = (f_1)) = \{f_1@E, f_1@C\} \tag{6.13}$$

$$L(f' = (f_2\ f_3)) = \{(f_2@C \mid f_3@C)\} \tag{6.14}$$

We use the placement sets to generate a cost graph $G_c = (V_c, E_c)$ (See Figure 6.5), where the vertices are the union of all placement sets: $V_c = \cup_{f'} L(f')$. The following are the values of the rest of the placement sets for the three functions example:

$$L(f' = (f_2)) = \{f_2@C\} \tag{6.15}$$

$$L(f' = (f_3)) = \{f_3@C\} \tag{6.16}$$

$$L(f' = (f_1\ f_2)) = \{(f_1@E \mid f_2@E), (f_1@C \mid f_2@C)\} \tag{6.17}$$

$$L(f' = (f_1\ f_2\ f_3)) = \{(f_1@E \mid f_2@E \mid f_3@E), (f_1@C \mid f_2@C \mid f_3@C)\} \tag{6.18}$$

We note that in order to try two memory configurations m_1 and m_2 on the cloud the placement set will have more combinations for example:

$$L(f' = (f_1\ f_2)) = \{(f_1@E \mid f_2@E), (f_1@C_{m_1} \mid f_2@C_{m_1}), (f_1@C_{m_2} \mid f_2@C_{m_2})\} \tag{6.19}$$

Step 3: Add cost graph links Once we have the nodes, as shown in Figure 6.5, we start by adding links from the *START* node to all nodes that start with first function in $FnSeq$, e.g. f_1. Similarly, we add links for all nodes that end with last function in $FnSeq$ (i.e., f_3), to the *END* node. To add links between intermediate nodes, we add links between each node that starts with function i, and each child node that starts with function j, where j is the successor of i in $FnSeq$. We discuss the details of calculating link weights in Section 6.4.2

Step 4: Solve the CSP problem: The constrained shortest path (CSP) problem is known to be NP-hard [96]. However, several approximation algorithms have been proposed to solve it [96][97][98]. Out of these methods, the LARAC algorithm [96] which is based on a relaxation of the CSP problem is an efficient algorithm to solve the problem. The main idea behind LARAC is to apply Dijkstra's shortest path algorithm on an aggregated cost $c_{uv}/c^* + \lambda d_{uv}/d^*$ that includes both the price and the delay values. The key issue in solving the CSP problem becomes how to

search for the optimal λ and determining the termination condition for the search. LARAC provides an efficient search procedure. We note that c_{uv} and d_{uv} are measured in different units ($ and seconds). Therefore, c_{uv} and d_{uv} are normalized through dividing them by c^* and d^* which are the maximum cost and delay values.

6.4.2 Cost Calculation

In this section, we show how we calculate the cost of the links in the cost graph. We take an example path:

$$(f_1@E) \rightarrow (f_2@C \mid f_3@C) \rightarrow (f_4@C) \rightarrow End \tag{6.20}$$

The path consists of 3 functions and the middle function is a fused function that consists of $f2$ and $f3$. We calculate both price and execution time cost for the link between each two consecutive functions. The price cost of the links on the path are added together to form the price cost of entire path. Similarly, the execution time of a path is the sum of executions times of the links on the path. In the following we focus on calculating the price P and the execution time T for each link and we follow the notation in Table 7.1

1. **Cost of link** $(f_1@E) \rightarrow (f_2@C \mid f_3@C)$:

$$T[(f_1@E) \rightarrow (f_2@C \mid f_3@C)] = \underbrace{e_{1,E}}_{\text{execution cost}} + \underbrace{tr(E \xrightarrow{D_{f_1}} C)}_{\text{transmission cost}} \tag{6.21}$$

$$P[(f_1@E) \rightarrow (f_2@C \mid f_3@C)] = \underbrace{p_E}_{\text{edge device price}} + \underbrace{r \cdot p_s}_{\text{transition price}} \tag{6.22}$$

2. **Cost of link** $(f_2@C \mid f_3@C) \rightarrow (f_4@C)$: We note that the cost of this link is different based on whether f_2 and f_3 are parallel functions or not, if they are parallel functions then the execution time will be bounded by the slowest functions, otherwise the execution time of the fused function will be the sum of execution time f_2 and f_3. The following is the calculation for both cases:

   If f_2 and f_3 are parallel:

$$T[(f_2@C \mid f_3@C) \rightarrow (f_4@C)] = \underbrace{max(s_{2,C} + e_{2,C}, s_{3,C} + e_{3,C})}_{\text{scheduling and execution time}} \tag{6.23}$$

$$P[(f_2@C \mid f_3@C) \rightarrow (f_4@C)] = \underbrace{r \cdot (e_{2,C} \cdot m_{2,C} \cdot p_{m_{2,C}}) + r \cdot (e_{3,C} \cdot m_{3,C} \cdot p_{m_{3,C}})}_{\text{functions price}} + \underbrace{2 \cdot r \cdot p_s}_{\text{transition price}} \tag{6.24}$$

if f_2 and f_3 are NOT parallel:

$$T[(f_2@C \mid f_3@C) \rightarrow (f_4@C)] = \underbrace{s_{2,C} + e_{2,C} + e_{3,C}}_{\text{scheduling and execution time}} \tag{6.25}$$

$$P[(f_2@C \mid f_3@C) \rightarrow (f_4@C)] = \underbrace{r \cdot (e_{2,C} + e_{3,C}) \cdot max(m_{2,C}, m_{3,C}) \cdot p_{max(m_{2,C}, m_{3,C})}}_{\text{fused function price}} + \underbrace{r \cdot p_s}_{\text{transition price}} \tag{6.26}$$

When f_2 and f_3 are not parallel, then they can be fused together which implies that: (1) they incur only one scheduling delay; (2) their execution times are added to each other; (3) the memory of the fused function is the maximum of the memory allocated to individual functions, and (4) they have one output transition.

3. **Cost of link** $(f_4@C) \rightarrow END$:

$$T[(f_4@C) \rightarrow END] = \underbrace{s_{4,C} + e_{4,C}}_{\text{scheduling and execution time}} \tag{6.27}$$

$$P[(f_4@C) \rightarrow END] = \underbrace{r \cdot (e_{4,C} \cdot m_{4,C} \cdot p_{m_{4,C}})}_{\text{functions price}} + \underbrace{\cdot r \cdot p_s}_{\text{transition price}} \tag{6.28}$$

6.4.3 Algorithm Analysis

Among the 4 steps described in section 6.4.1, solving the constrained shortest path (CSP) problem (Step 4) is the one that dominates the algorithm complexity. The complexity of LARAC's algorithm [99] for solving CSP on the cost graph is given by:

$$O(\mid E^2 \mid log^2(\mid E \mid)) \tag{6.29}$$

such that $\mid E \mid$ is the number of links in the cost graph which is defined as.

$$| E |=| V | \cdot deg_v \tag{6.30}$$

such that $| V |$ is number of vertices in the cost graph, deg_v is the maximum out degree of each vertex. In order to calculate these values, we denote R as the number of devices in which the function can be placed on, where $R = 2$ for one edge device and one cloud configuration, and $R = 4$ for one edge and 3 cloud configurations. $| V_p |$ is calculated as the number of fused functions multiplied by the number of possible placements (i.e., R) of each fused function (See Step 2 in section 6.4.1). Given n functions, the number of fused functions of size k is given by $(n-k+1)$, therefore the total number of vertices $| V |$ in the cost graph can be written as:

$$| V |= \sum_{k=1}^{n} (n-k+1) \cdot R \tag{6.31}$$

The worst case out degree of cost graph nodes is:

$$deg_v = (n-1) \cdot R \tag{6.32}$$

This happens for nodes $f1@C$ and $f1@E$ which are connected to $(n-1)$ fused functions that starts with $f2$ and for each fused function there are m nodes that represent each possible placement of the fused function. By substituting Equations 6.31 and 6.32 in Equation 6.30, the number of links can be defined as:

$$\begin{aligned} | E |= R^2 \cdot (n-1) \cdot \sum_{k=1}^{n} (n-k+1) = R^2 \cdot (n-1)(\sum_{k=1}^{n} n - \sum_{k=1}^{n} k \\ + \sum_{k=1}^{n} 1) = R^2 \cdot (n-1)(n^2 - (n^2+n)/2 + n) = O(R^2 \cdot n^3) \end{aligned} \tag{6.33}$$

By substituting Equation 6.33 in 6.29, the overall complexity is given by:

$$O(| E^2 | log^2(| E |)) = O(R^4 \cdot n^6 \cdot log^2(R^2 \cdot n^3)) \tag{6.34}$$

6.5 EVALUATION

Experimental Setup: The computing infrastructure consists of edge and cloud resources. We use the Raspberry Pi Model B as the edge device. For the cloud side we use AWS Lambda and AWS Step Functions to create a workflow of lambda functions. We set the default memory of each

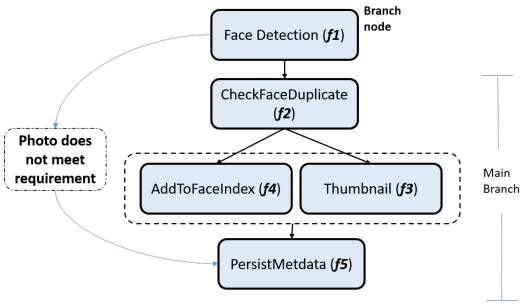

Figure 6.7: Wild Rydes application workflow

lambda function according to the closest values to the maximum memory used by the function. For example, if the function profiling shows that a function uses a maximum of 100 MB of memory, we allocate memory to be the closest allowed value by AWS Lambda which is 128MB. We set the timeout to a large value to keep the function running until it finishes execution. We assume that the data comes from the Raspberry Pi and it is uploaded to AWS Storage (S3) for cloud processing. Once the data is uploaded to S4, it automatically triggers the execution on the cloud. If a function is placed in the edge, it is executed first and the intermediate data is transmitted to the cloud.

Application: We evaluate the performance of our algorithm using Wild Rydes application workflow [100]. Wild Rydes is a transportation application similar to Uber that allows users to request rides in an on-demand manner and the application matches them with the closest drivers. Wild Rydes requests its users to upload their photo when they sign up for a new account. Once the user uploads their photo, the image processing workflow in Figure 6.7 starts executing. The workflow consists of five functions that process the image, matching it across a database of faces and indexing the uploaded face for future matching. The workflow is implemented in *JavaScript* and it takes an image as input. For the sake of brevity, we label the functions from $f1$ to $f5$ as shown in Figure 6.7 and we use the labels for the rest of the chapter. $f1$ is a branch function that decides the execution of the rest of the application. As described in section 6.4.1, we do not include the branch function $f1$ in the fusion and we explore fusion for the rest of the 4 functions. $f1$ is a face detection function that uses AWS Rekognition library which is a cloud-based library offered by Amazon that contains a variety of image and video processing functions. Since AWS Rekognition does not offer a distribution that can be deployed on a Raspberry Pi, we implemented the face detection functionality using Python's Dlib [101] library. The rest of the functions is implemented on the cloud because the functions need to access two databases of faces and metadata that are stored on the cloud.

Application Profiling: We profile the application by executing the workflow for 20 times on both the edge and the cloud. On the edge, we run the *FaceDetection* only because it is only function that does not depend on the cloud database. On the cloud we run each function with

Function	Avg. exec. time [128 MB / 256 MB / Edge]	Avg. scheduling delay	Max Memory used	Avg. billed duration [128 MB / 256 MB]
f1	893 ms / 772 ms / 1870 ms	61 ms	42 MB	955 ms / 822 ms
f2	970 ms / 743 ms	52 ms	38 MB	1016 ms / 800 ms
f3	2063 ms / 1080 ms	172 ms	83 MB	2116 ms/1144 ms
f4	844 ms / 735 ms	153 ms	37 MB	883 ms/788 ms
f5	153 ms / 101 ms	67 ms	38 MB	211 ms/144 ms

Table 6.2: Profiling information for the functions in the Wild Rydes application (Figure 6.7).

two memory configurations $m_1 = 128$ and $m_2 = 256$. We use AWS logs to extract the following profiling information **for each function**:

1. Average Execution time on the cloud using 128 MB (Default configuration unless otherwise stated)
2. Average execution time on the cloud using 256 MB
3. Average execution time on the edge
4. Average scheduling delay on the cloud
5. Maximum memory used
6. Average billed duration using 128 MB
7. Average billed duration using 256 MB

Table 6.2 shows the results of the profiling. We note that the average billed duration is always greater than the average execution time because it is rounded up to the nearest 100 milliseconds. For example if the execution is 720ms, AWS charges for 800ms so the average billed duration tends to be bigger than the average execution. We also notice that the first run is usually much slower than other runs. This is probably due to the fact that AWS provisions a container in the first run that can be reused for other requests. We consider the first run as a warm up run and we do not include it in the profiling.

We further run some benchmarking to measure the transmission time from the Raspberry Pi to the Amazon S3 and we measure the duration from sending the 100 images to receiving a response from S3 that the data has been uploaded successfully. We use this information to measure the speed of transfer from the edge to the cloud and on average it takes 1.13 seconds to upload a 720p image with sizes between 1.2-1.5 MB size. We note that uploading the image is also needed when $f1$ is executed on the edge because the image is needed by other downstream tasks.

Evaluation Metrics: We conduct experiments to evaluate Costless performance in terms of:

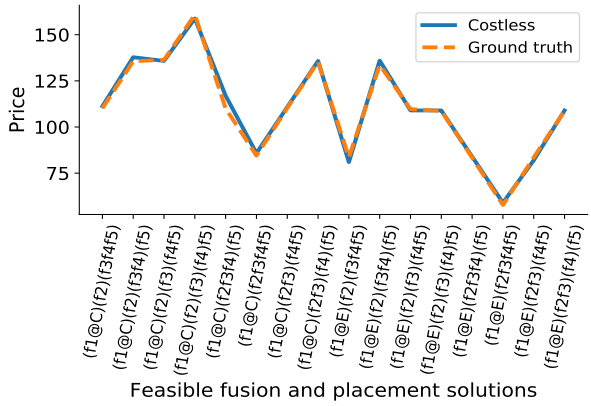

Figure 6.8: Comparing pricing estimates of Costless with observed times during deployment of manually fused functions

- *Model Accuracy:* For each fusion and placement solution, we compare our estimate of price and execution time to the observed completion time when we manually fuse and place functions, and we evaluate accuracy of our estimates compared to the billing information from AWS.
- *Price within latency constraint*: Given a function graph and some deadline, we use Costless to find the best price for the deadline. We compare the results of Costless with Brute force solution and other heuristics.
- *Effect of optimizing over memory configurations*: For each fusion and placement solution, we show the price optimization when we search over different memory configurations for each function.
- *Time to find Solution*: We measure the time that Costless takes to obtain the placement and fusion decision and we focus on its behavior for large scale function graphs.

6.5.1 Model Accuracy

In order to show that our modeling of price and execution closely reflects the observed running time from manually fused functions, we create all feasible function fusions manually for the application in Figure 6.7. For the 4 functions in the workflow, we try all the combinations of placements and fusions (e.g., fuse two, three or four consecutive functions). We run the manually fused application on AWS and we use logs to find the actual price and execution time and we use this as our *Ground truth*. We note that some fusions made the two parallel functions *AddToFaceIndex* and *Thumbnail* run sequentially. Figures 6.8 and 6.9 show the results for the execution time and

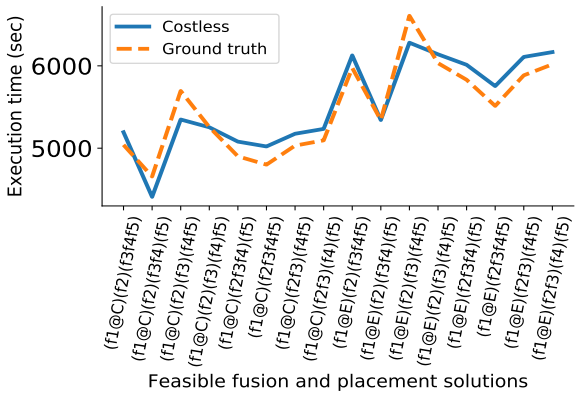

Figure 6.9: Comparing execution time estimate of Costless with observed times during deployment of manually fused functions

price estimates. Figure 6.8 shows that the estimated price and execution for each of the possible solution were very close to the empirically measured values. The execution times in Figure 6.9 shows slight discrepancies compared to the ground truth obtained from AWS logs, this is due to the fact that there is a scheduling delay between the time of receiving the request and the time at which the function starts execution. Such delay is highly variant from one request to another and it causes our execution time estimate to deviate by 100-300ms. The execution time estimate, however, follows the same trend as the ground truth and it could capture all the peaks. The average error was only 1.2% for price and 4% for execution time.

6.5.2 Price vs. Execution Time

In this experiment, we show the relationship between the price and execution time of feasible fusion and placement solutions obtained by Costless. Figure 6.10 depicts the price execution time relationship. As shown in Figure 6.10, there is no clear trend that when the price increases the execution time decreases. In fact, different fusions can give different prices and for each price point there can be different execution times. This due to the fact that some fusions can cause two parallel functions to run sequentially so the price decreases but the execution time increases. In Figure 6.10, we focus on four data points that are the most interesting because they have the smallest execution times but yet they have different prices. We note that the most expensive is the original graph in Figure 6.7 and thats because it has four transitions apart from the start and end transition that are common in all solutions. Though it is the most expensive, it has the best execution time because it can leverage the parallelism between the two functions $f3$ and $f4$. On the other hand the cheapest solution is to place $f1$ on the edge and fuse the functions $f2f3f4f5$,

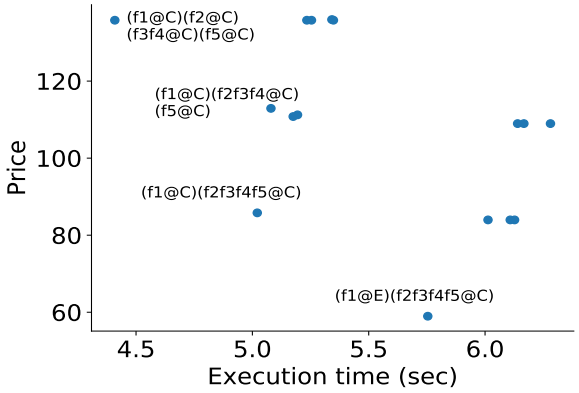

Figure 6.10: Time and Price estimate for each feasible fusion and placement solution obtained by Costless

fusing the functions not only decreases the price but it also eliminates the need for a scheduling delay between consecutive functions because now several functions became just one function with one scheduling interval so the application runs faster. There are two points in between the cheapest and the most expensive, one of them is very similar to the cheapest solution but it does not use edge device and the other one is more expensive because it fuses 3 functions instead of 4. We note these intermediate solutions that Costless quantifies are helpful because they tend to fuse less so the application retains some of its modularity while improving the price.

6.5.3 Price within Latency Constraint

In this section we show the benefits of Costless to optimize the price within latency constraint. The goal is that for some deadline (e.g., 5 seconds), we need to find the solution with the lowest price. We compare Costless with several heuristics to assess its ability to accurately find the lowest price solution and we also show it improves on simple decisions that supports using cloud only or edge only. The heuristics compared are:

- Ground truth: these are the results obtained from manually fusing the functions and running them on AWS. The logs of AWS are automatically parsed to find out the lowest price for a given deadline.
- Costless: this is the approach presented in this chapter and it uses the price and execution time estimates described in section 6.3 and the algorithm described in section 6.4.1
- Bruteforce: this approach uses the price and execution estimate described in section 6.3 but instead of constructing a cost graph and using the constrained shortest path approximate

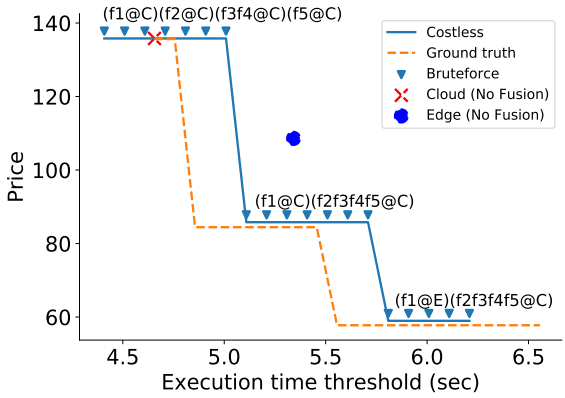

Figure 6.11: Best price below an execution time threshold.

algorithm, it searches over all the solutions in a brute force fashion.

- Cloud (No Fusion): Keep the original application with no fusions and place all the functions in the cloud
- Edge (No Fusion): Keep the original application with no fusions and place functions on the edge device whenever possible

Figure 6.11 shows the results for finding the price for each execution time threshold. The results shows that even though Costless uses an approximate algorithm to solve the constrained shortest path problem, the solutions it found exactly matches the solutions found by Brute force. We also note that the solutions found by Costless are close to the Ground truth obtained from AWS logs, except that Costless sometimes switches to a different price slightly later than the ground truth. We attribute this to the same reason we described in Figure 6.9 in which the time estimate of Costless is slightly different than the ground truth. However, we can see that Costless eventually reaches the same price values obtained from AWS logs and the places where a mismatch occurs is only within 200-300ms. We further show that the simple policies: *Cloud (No Fusion)*, *Edge (No fusion)* misses the opportunity to reduce the price with a small difference in execution time, for example, Costless can reduce the cost by 37% (135$ - 85$) with only 5% increase in latency. The best price Costless can achieve is 58$ which is 57% reduction with 15% increase in latency.

6.5.4 Effect of Optimizing over Memory Configurations

In this section, we show how changing the memory configuration can optimize the price. Using the profiling information in Table 6.2 we were able to efficiently search over all the combinations of fusion and memory configurations the same way we search over placement of functions on edge

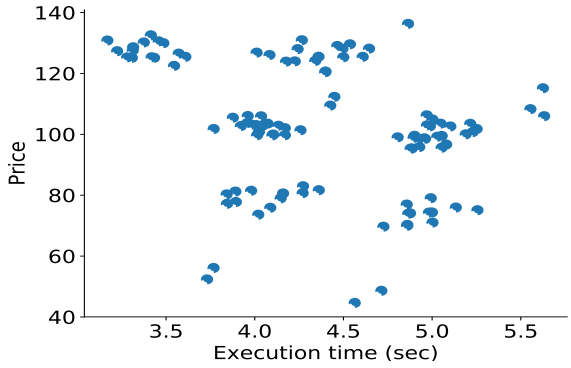

Figure 6.12: Price and time for different fusions and memory configurations

and cloud. For each fusion solution, we search over two different memory configurations (128 MB and 256 MB) for each function. We note that for the non-fused case, we have 5 functions so there are 25 combinations but after fusing we might end up with 2 functions so we search for 4 combinations. Figure 6.12 shows the relationship between the execution time and the price for different solutions of fusion and memory configuration. The Figure shows that the data points are organized into clusters. Each cluster represents one fusion solution and within each cluster there are several data points that have different prices and execution times. We note that for the non-fused case, the solution that have the best price was non-trivial since it keeps the memory for the 4 functions at $128MB$ and it increases the memory of $f3$ only to $256MB$, this ended up being the best solution because $f3$ experienced the highest speedup when the memory increased from 128 to 256 (See Table 6.2). Such speedup has a positive impact on the overall price and execution time. We conclude that setting the memory configurations manually is not ideal and running all profiling configurations is very expensive. On the other hand, profiling each function separately and running a scalable algorithm such as Costless can help finding a non trivial and cheaper memory configurations. The best solution we found improved both the price and the execution time by 6% and 10% respectively.

6.5.5 Time to find Solution

In this section, we evaluate the scalability of Costless with increasing the number of functions. We generate synthetic functions and we append it to the end of the graph. For each new function we randomly sample its profile which include (1) execution time on the cloud (500s-2s), (2) execution time on edge (1s-5s), and (3) scheduling delay (50ms-300ms). Figure 6.13 compares Costless with Bruteforce search, in the time to obtain the placement. The time for Costless includes the construction time of placement graph and executing constrained shortest path algorithm. The

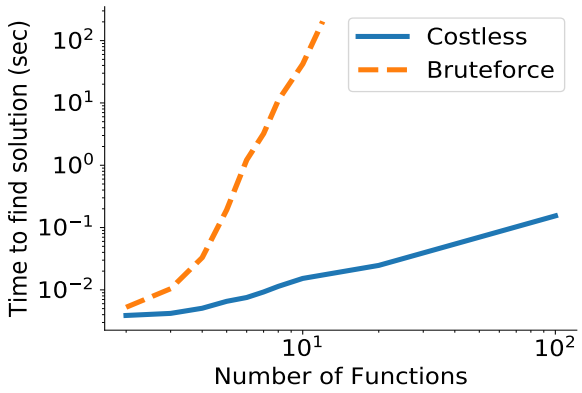

Figure 6.13: Comparison between Costless and brute force in the time in the time to search for the best solution with increasing number of functions

brute force time includes calculating the cost for each feasible solution, sorting them based on the price, and finding the lowest price that have latency above certain threshold. With Bruteforce, we were not able to complete the search in a reasonable amount of time when number of functions increased beyond 12. Costless however was able to complete the search within one seconds for 100 functions. The scalability of Costless comes from two main factors: (1) Constructing the cost graph which avoids redundantly computing the cost of fused functions that are shared between multiple solutions, (2) Formulating the problem as a constrained shortest path problem which allows Costless to use scalable heuristics.

6.6 CONCLUSION

In this chapter, we studied the problem of optimizing the price and execution time for serverless computing. We identified three fundamental factors affecting the price of serverless applications which are: function fusion, function placement, and memory configuration of serverless functions. Our fundamental idea was to represent fusion and placement solutions in one cost graph and we presented an efficient algorithm to obtain the best solution given latency or price constraints. Although function fusion has the disadvantage of making the application less modular and maintainable, we showed that it was an effective way to reduce the cost, especially when transition cost dominates the function execution cost. We were able to reduce the price of an image processing application by more than 37% with 5% increase in the latency and we showed that placement of functions on edge devices can help increase the price reduction to 57% . We also showed that using the right memory configuration can help reduce both price and latency of the application deployment.

CHAPTER 7: SERDAB: NEURAL NETWORK PARTITIONING ACROSS MULTIPLE ENCLAVES

The usage of public cloud infrasturcutre comes with more challenges other than the price optimization described in Chapter 6. Protecting the privacy and confidentiality of users' sensitive data against misuse by the edge/cloud provider is another major challenge that we address in this chapter. Trusted Execution Environments -also known as TEEs or enclaves- such as Intel Software Guard Extensions (SGX) have emerged as a prominenent solution to run machine learning workloads in public clouds while preserving data confidentiality. In such systems, the private data is only decrypted within the trusted execution environment which is protected from all privileged software in the system such as operating system and virtual machine monitors. However, SGX-based computation is currently performance-and memory-constrained. For example, TEE workloads cannot exploit accelerated linear algebra libraries. It also has a limited memory size (128 MB) which limits the amount of computation that can be done within one TEE and it becomes essential to delegate part of the computation to run in another hardware accelerator (e.g., GPU, CPU, or another enclave) that sits in the same or a different edge device. Hence in this chapter, we extend our distributed orchestration framework described in Section 3. Our extended orchestration framework, Serdab[1], simplifies the deployment, and management of visual IoT applications, seamlessly across multiple enclaves that sit in different devices. Serdab also includes a novel technique to find the best partitioning of neural networks computation across distributed resources.

The rest of the chapter is organized as follows. Section 7.1 discusses background, motivation, and the threat model. In Section 6.2, we present an overview of the optimization goals and the tradeoffs that we address in this chapter. In section 7.3, we describe the system architecture. We define the problem of privacy-aware placement in section 7.4 and we present our technique to solve it in section 7.5. We evaluate our system in section 7.6. Section 7.8 concludes the chapter.

7.1 BACKGROUND AND MOTIVATION

7.1.1 Trusted Execution Environment (TEE)

Trusted Execution Environment such as Software Guard Extensions (SGX [26]) is available on Intel processors starting with Skylake. TEE provides a secure area in the processor that protects code and data from all other privileged software on the platform. The code in the TEE is executed

[1] An ancient Egyptian tomb structure in which secret chambers (enclaves) are connected with secret passages (communication channels)

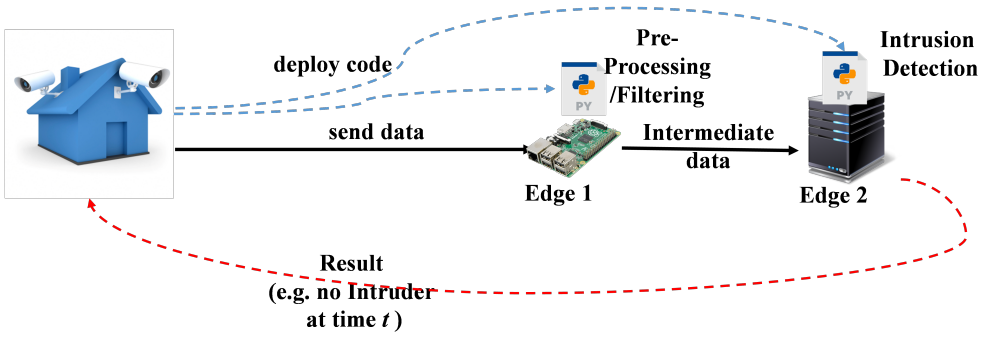

Figure 7.1: Intrusion detection application using security cameras. Example application that benefits from processing in the TEE.

safely on secret data that nobody outside the TEE can have access to it including the hardware vendor (e.g., Intel). The privacy and integrity of the code/data inside the TEE is enforced by the hardware. TEEs are sometimes called enclaves, and we use both terms interchangeably in this chapter. Intel SGX supports remote attestation of the code and data in the TEE [27]. This enables a remote user to verify the trustworthiness of the hardware and the integrity of the TEE contents (i.e., code and data). The size of the memory reserved to the TEE is limited to 128 MB. However, SGX supports *paging* in which the rarely used Enclave Page Cache (EPC) pages are evicted to the unprotected main memory, but they remain encrypted to ensure confidentiality. With paging, applications in TEE can use more than 128 MB at the expense of encrypting and decrypting the evicted pages which poses an additional efficiency challenge.

7.1.2 Example Application and Threat Model

Figure 7.1 shows an example application of intrusion detection. In this application, the user installs cameras to monitor the area around his house, and he wants to run computer vision software that analyzes the videos from the installed cameras to detect threats like: (1) the presence of intruders in the backyard, (2) car break-ins, (3) package theft. Many users/application developers do not have the expertise/compute resources to deploy such software services in their houses, so they utilize services provided by public cloud providers (e.g., AWS) or edge providers (e.g., VaporIO). However, there are several key requirements that users need to consider in order to trust such services:

- **Code Integrity:** Assuming that the user knows the code/deep learning model he/she wants to run on the data, how to ensure that service provider is running the exact code that the user (app developer) submitted remains a concern. Malicious/Compromised service providers can drop the code and return random results or can inject code that makes copies of private

user data.

- **Data Privacy:** Even if the code remains unchanged by the cloud/edge service provider, users want to ensure that they get the right predictions without their data being revealed to the service provider. User's data can contain private information about the activities they do around the house, the cars they own, and the guests coming to visit them. Users want to ensure that the data cannot be seen in an unencrypted form except by the software that analyzes the data.

Threat Model: The security objective of Serdab is to protect the privacy of user inputs (e.g., home camera feed) that are being used by deep learning inference services hosted by third-party edge/cloud providers. There are four parties in our system: (1) The user that owns the video data (e.g., homeowner), (2) The App developer that developed the Deep Neural Network (DNN) inference service, (3) Serdab framework that deploys the DNN inference service in the edge devices or cloud platform, and (4) The edge/cloud provider that manages the hardware resources that execute the inference on the user's data. The user trusts the app developer in providing a privacy-preserving DNN inference. The user and the App developer do not trust Serdab framework to deploy the required services in the Intel SGX devices supported by the cloud provider. However, both the user and the App developer have a method, provided by Intel [27], to perform remote attestation on all the trusted hardware that they rent to ensure that the code has actually been deployed by Serdab. The edge/cloud provider is the adversary in our threat model, and its goal is to steal user inputs but provide correct output. We assume that the edge/cloud provider has no incentive to produce false outputs, but they have incentive to keep the user's data for use in other activities that are not explicitly approved by the user such as Targeted Advertisements [102]. This is now common that an attacker wants to use data for their economic benefit so they will not damage the system and quietly leak the data. In our setup, the edge/cloud provider administrators are capable of accessing on-cloud software and hardware resources except the SGX's enclaves. The communication channel from the user's cameras to the enclave and between enclaves is protected by TLS or similar secure protocols. We note that although the data transmission between enclaves has to happen through the untrusted hardware, users can attest that each enclave encrypts its output before being transmitted to the next enclave. We do not consider the SGX side-channel attacks [103], which can be prevented [104], as well as Denial of Services (DoS). We assume that the DNN models to be deployed in the cloud are trained in a secured environment, the model parameters are not leaked to adversaries during model training, and the user's camera(s) are not compromised by adversaries.

7.2 OPTIMIZATION GOALS

In this section we present an overview of: (1) The utility that we aim to optimize, (2) the applications that we addres, (3) the underlying compute resources that we levarage, and (4) the tradeoffs that we explore:

Utility: We optimize the data and application's privacy when processing a stream of data by an analytics application.

Application: A sequence/graph of neural network layers.

Compute resources: Multiple Edge devices owned and managed by public edge/cloud provider. The devices are equiped with trusted execution environments (TEEs).

Tradeoffs: We aim to control the tradeoff between the application's privacy, the application's end-to-end latency, and the application's locality (i.e., the ability to process the entire application closer to where it is generated)

7.3 SYSTEM OVERVIEW

To address the requirements specified in Section 7.1, we propose a framework to support distributed analysis of IoT data streams across multiple enclave devices. We focus on visual IoT streams which come from cameras owned by users/organizations (e.g., surveillance cameras, home cameras, wearable cameras, etc). An architectural overview of Serdab framework is presented in Figure 7.2. The top component is the application layer which defines an application in the form of a Directed Acyclic Graph (DAG) of functions/operators. In this chapter, we focus on DNN applications in which the application is a NN model, defined as a DAG of NN layers $\{L_x\}_{x=1}^{x=M}$. Each layer L_x represents a compute operation. The layers in a DNN model include: convolutional layers (that combine nearby pixels via convolution operators), pooling layers (that reduce the dimensionality of the subsequent layers), rectified linear unit (ReLU) layers (that perform a non-linear transformation), and fully connected layers (that perform matrix addition, multiplications). The framework manages multiple devices that could be connected either via local area network or a wide area network. In Figure 7.2, we show two devices (edge 1 and edge 2) at different locations connected via a wide area network and each device has a trusted enclave.

Edge-Cloud Orchestration: The second component from the top in Figure 7.2 is an edge-cloud orchestration framework which provides an abstraction for deployment and management of NN layers across edge 1 and edge 2 dataflow engines. The orchestration engine is a control component that can be deployed in edge 1 or edge 2 or another device controlled by the user but connected to edge 1 and edge 2 via a local or wide area network. The orchestration engine has a *Resource Manager* which carries information about which compute resources are available to execute the

NN model (e.g., edge 1 and edge 2). We assume that the edge/cloud provider reports the available resources (devices) correctly. The Resource Manager receives requests to dynamically register new resources and removes old ones. The orchestration framework includes an *Application Manager* that sends a request to the local dataflow engine of both edge 1 and edge 2 to deploy the subset of the layers assigned to each device. The *Application Manager* also allows encrypted data to flow from one Data-flow Engine to another by deploying a *transmission operator* in each Data-flow Engine. The transmission operator pushes the output of the final layer in one Data-flow Engine to the first layer in the next data-flow engine.

Privacy-aware Placement: Before the Application manager enacts the deployment, it consults the privacy-aware placement service to find the best placement of layers across devices. The main goal of the privacy-aware placement is to consider the available resources (i.e., edge/cloud devices) to improve the latency of the application without violating the data privacy. The privacy-aware placement tries to reduce the amount of layers computed with one TEE through offloading the rest of the layers to another untrusted device or another enclave. However, the privacy-aware placement has to ensure that the output of the layers coming out of one enclave to an untrusted device is *sufficiently dissimilar* to the original image. We provide more details about the privacy-aware placement and our definition of image similarity in Section 7.5.

Dataflow Engine: Below the edge-cloud orchestration workflow is the set of dataflow engines. Each edge device has a local stream processing engine that handles the execution of the sub-DAG of layers that are deployed in it. The dataflow engine plays a management role within one device, particularly it handles the dataflow between the trusted and untrusted hardware in the same device. Each operator in the dataflow engine acts as a client to a service that is implemented in the trusted hardware or a regular CPU. The operator calls the NN Inference service and passes the encrypted data to it. The NN inference will be described in detail in the next section. The calling operator gets back the encrypted result to forward to the next operator. Some operators are transmission operators that do not talk to any service but instead they forward the data over the wide area network to the data-flow engine in another edge device.

NN Inference Service: The NN inference service is implemented as a gRPC service enclosed within a Docker container. When the service is initially deployed, Serdab informs the user to upload the encrypted model parameters directly to the enclave service. The encrypted model parameters will only contain the layers that this enclave is supposed to serve. Once the model parameters are loaded, the gRPC service starts receiving encrypted video frames and executes the inference through Tensorflow Lite (TFlite) library which is a lightweight implementation of TensorFlow [105] for resource-constrained devices such as SGX.

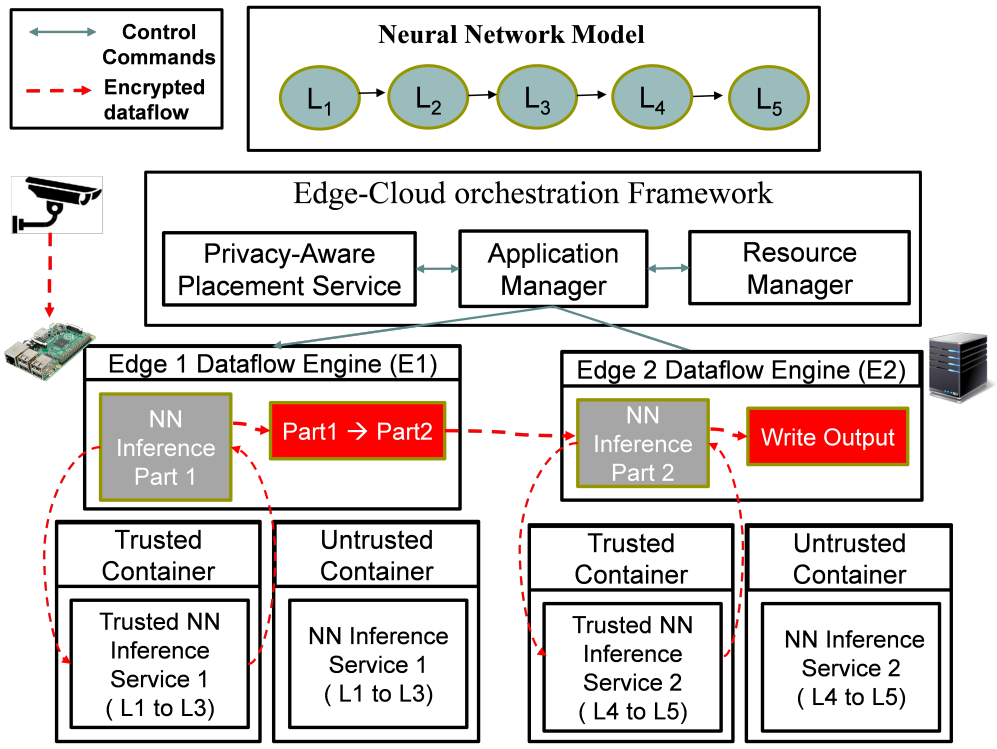

Figure 7.2: Serdab System Architecture.

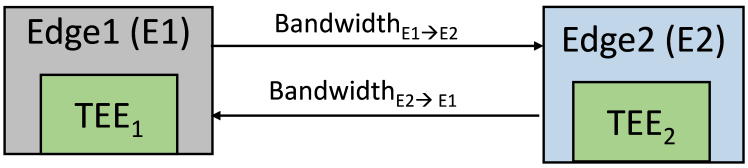

Figure 7.3: Example Resource graph

7.4 MODELS AND PROBLEM DEFINITION

The performance and memory limitations of TEE (e.g., 128 MB in SGX) make it in-adequate to efficiently support running an entire deep and complex neural network model. Hence, privacy-aware placement aims to explore different ways in which some layers of the neural network are offloaded from a TEE to another device or another TEE with the aim to improve latency while ensuring that the data remains private. In this section, we define the problem of privacy-aware placement. We start by describing the NN model, compute resource model, and IoT data model. Then we formally define the problem. In Section 7.5, we present the intuition behind our approach and the algorithm to solve it. In both sections, we follow the notation in Table 7.1.

Resource Model: We model the physical resources as a weighted directed graph $G_R = (V_R, E_R)$, as shown in Figure 7.3, where the vertices $V_R = \{E_1, E_2, TEE_1, TEE_2\}$ represent two edge devices

M	=	Number of layers in a neural network
NN	=	Neural network model with M layers
TEE_l	=	Trusted compute Resource
E_l	=	Untrusted compute Resource
V_R	=	All Compute resources (e.g., TEE_1, E_1)
V_{R_T}	=	Trusted resources (e.g., TEE_1, TEE_2)
$V_{R_{UT}}$	=	Untrusted resources (e.g., E_1, E_2
L_x	=	Layer x in a neural network
e_{x,E_1}	=	Execution time of L_x in E_1
n	=	Total number of frames in a chunk
$L_x@E_1$	=	Placement of layer L_x in E_1
$L_x \rightarrow L_z@E_1$	=	Placement of layer L_x to L_z in E_1
p_x	=	Resource (device) that L_x is placed on
P_j	=	defines a placement path j for each layer in NN to a resource in V_R
L_x	=	Layer x in a neural network
$t_{chunk}(n,P_j)$	=	Completion time of a chunk of n frames given placement P_j
f_y	=	Frame y in a chunk of frames
$I(L_x)_y$	=	Input to L_x when the input image is f_y
$Sim(I(L_1),I(L_x))$	=	Similarity bet. input to L_x and input to L_1
δ	=	Threshold on the similarity between 2 layers' inputs
D_{L_x}	=	Size (bytes) of output tensor of layer L_x
B_{E_1,E_2}	=	Bandwidth (bytes/sec) bet. E_1 and E_2
$tr(E_1 \xrightarrow{D_{L_x}} E_2)$	=	Transmission time (sec) bet. E_1 and E_2

Table 7.1: Table of Notation

E_1 and E_2. Each device can have a trusted execution environment (TEE). We refer to the TEE inside E_1 as TEE_1, similarly we refer to the TEE inside E_2 as TEE_2. We divide the resources (devices) into two distinct sets: trusted resources V_{R_T} and untrusted resources $V_{R_{UT}}$, where $V_{R_T} = \{TEE_1, TEE_2\}$, and $V_{R_{UT}} = \{E_1, E_2\}$. The links $E_R = \{(B_{E_1 \rightarrow E_2}, B_{E_2 \rightarrow E_1})\}$ represent bandwidth availability from E_1 to E_2 and vice versa.

IoT Data Model: We model the data as an unbounded stream of video frames f_y that can come from one or more camera sources. We aggregate a sequence of video frames into *chunks*. The k^{th} chunk $chunk_k$ can be defined as $chunk_k = \langle f_1^k, f_2^k, \ldots f_n^k \rangle$, where n is the chunk size (i.e., number of video frames). The chunk size is an application-defined parameter to define how often the partitioning algorithm gets invoked to dynamically change the partitioning.

Application Model: As described in Figure 7.2, we model the application as a NN model that has multiple layers $NN = \{L_x\}_{x=1}^{x=M}$. Each layer is a processing element such as convolution or matrix

multiplication. A link from layer L_1 to layer L_2 means that for a given video frame f_y, L_1 has to be applied before L_2 and the output of L_1 is the input for L_2.

NN Layer Profile: Each layer in the NN is associated with a *profile* which includes:

1. The cost of executing layer L_x on a video frame f_y, when placed on each compute resource. Layer L_x can be placed on E_1, TEE_1, E_2, or TEE_2. Their corresponding execution times are e_{x,E_1}, e_{x,TEE_1}, e_{x,E_2}, and e_{x,E_2}. The execution includes the time to encrypt the layer's output.

2. The size of output data of layer L_x. We denote it as D_{L_x} bytes. The output data of each layer is typically n-dimensional matrix (i.e., tensor) and its size can be derived from the resolution of the input frame.

3. The transmission time of D_{L_x} from E_1 to E_2, $tr(E_1 \xrightarrow{D_{L_x}} E_2) = D_{L_x}/B_{E_1,E_2}$, where B_{E_1,E_2} is the bandwidth (bytes/sec) from E_1 to E_2.

4. The similarity between the input to layer L_x, denoted as $I(L_x)$, and the original image f, which is also the input to the first layer $I(L_1)$. The similarity is defined by a similarity function $Sim(I(L_1),I(L_x))$ (e.g., Pearson correlation coefficient, mean-squared error). This similarity metric represents how much a layer leaks information about the original image. To compute the similarity, we use a dataset of 1000 diverse images and we get the intermediate output of each layer. For each image f_y, we run similarity function between $I(L_1)_y$ and $I(L_x)_y$, where $I(L_x)_y$ is the input to L_x when the input image is f_y. We compute the overall similarity of the layer as the maximum across all the images: $Sim(I(L_1),I(L_x)) = max_y(Sim(f_y,I(L_x)_y))$.

Problem Definition: Consider a NN model $NN = \{L_x\}_{x=1}^{x=M}$, which consists of M layers. Let $V_R = V_{R_T} \cup V_{R_{UT}}$ be the list of available resources to execute the layers including the list of trusted resources V_{R_T} and untrusted resources $V_{R_{UT}}$. Let p_x denote the placement of an arbitrary layer L_x on the compute resources that it will be executed on $p_x = r_k \mid r_k \in V_R$. A placement path $P_j = (p_1, p_2, ..., p_M)$ defines a placement for each layer in NN to a resource (device) in V_R. We note that P_j defines one placement out of combinatorial number of placement of layers in NN on resources V_R. For example, a simple placement $P_1 = (TEE_1, TEE_1, ...TEE_1, TEE_1)$ denotes that all the layers are placed on TEE_1 while $P_2 = (TEE_1, , ..., TEE_2, TEE_2)$ denotes that some layers are placed on TEE_1 and some are placed on TEE_2. Let n denote the number of video frames (i.e., chunk size) processed by NN and let $t_{chunk}(n, P_j)$ denote the time it takes to perform a complete execution of NN on n video frames given a placement path P_j. The goal of the privacy-aware placement is to find the path P_j^* that minimizes the execution time $t_{chunk}(n, P_j)$:

Figure 7.4: Output of intermediate layers of GoogLenet: left (original image), middle (layer 1: maxpool), right (layer 4: maxpool)

$$P_j^* = \underset{P_j}{\operatorname{argmin}}\left(t_{chunk}\left(n, P_j\right)\right) \tag{7.1}$$

while satisfying at least one of the following privacy constraints:

C1: Each layer is executed within a trusted device.

$$\forall p_x \in P_j \ , p_x \in V_{R_T} \tag{7.2}$$

C2: If a layer is executed in an untrusted device, then the input to that layer has to be sufficiently dissimilar to the input of the first layer of the original image ($I(L_1)$). The similarity is defined by the similarity function and a threshold δ.

$$\forall p_x \in P_j \ if \ p_x \in V_{R_{UT}}, \ then \ Sim[I(L_x), I(L_1)] < \delta \tag{7.3}$$

7.5 APPROACH

Our privacy-aware placement technique explores alternative placement that can potentially improve the overall latency of the NN application without violating the privacy. Our privacy-aware placement method relies on two **key insights**:

- **dissimilar intermediate outputs:** Several studies have been conducted to understand the internal mechanisms of DNNs [81]. Such studies have shown that for an image classification DNN, the first few layers (shallow layers) represent low-level image processing operations such as edges, corners, and contours of the original inputs. On the other hand, later layers (i.e., deep layers) represent more abstract and class-specific information related to the final

outputs. Figure 7.4 shows an illustrative example for the output of layer 1 and layer 4 of Google's InceptionNet (GoogLenet). As shown in Figure 7.4, layer 1 does primitive image processing operations (e.g., edge detection) so the content of the image is still visually identifiable by humans acting as adversaries. However, the output of layer 4 is significantly less similar to the original image. This insight can be used in the context of TEE to execute the first 4 layers inside the enclave until the image becomes dissimilar to the original image. The rest of the layers (after layer 4) can then be executed in a regular processor to leverage hardware accelerators such as GPUs. However, the question becomes what is the minimum number of layers to be executed within the resource-constrained TEE that is enough to conceal the identity of the data. To answer this question, we measure the correlation between the original input image and each layer's output. After experimenting with several correlation metrics (e.g., mean-squared-error (MSE), Pearson correlation coefficient, and structural image similarity (SSIM)) and conducting a user study, we realize that the most crucial metric that affects the correlation between the input image and the intermediate layer is the *resolution* (i.e., number of pixels) of intermediate layer's output. We notice that the output of each layer has a grid of images (Figure 7.4), where each image in the grid is either smaller than or equal to the images in the input grid. This happens because convolution and pooling operations reduce the number of pixels in each image in the grid. We observe that when the output of a layer has images with resolution less than or equal a certain threshold (e.g., 20x20 pixels) then such output has undergone enough transformations in which it cannot be visually identified no matter how much you can resize it. Hence, we conclude that the output of layer L_x is considered private if the resolution of its output is below certain threshold δ. We note that our privacy-placement method is not restricted to using the resolution as a metric and more complex similarity metrics can be utilized.

- **pipeline parallelism:** A key idea in our privacy-aware placement approach is to allow processing a stream of video frames in a pipeline fashion. Pipelining allows utilizing both TEE_1 and TEE_2. For example, while the TEE_1 is processing the first part of the neural network for the second video frame, TEE_2 will be processing the second part of the neural network for the previous frame. Figure 7.5 shows the execution and transmission time for executing a neural network in three different cases: (1) all layers of the NN are deployed on the TEE_1 (left column), (2) the NN is partitioned across TEE_1 and E_2 (middle column), and (3) the NN is partitioned across TEE_1 and TEE_2 (right column). Figure 7.5 shows that the best completion time for one frame is the second case when NN is partitioned across TEE_2 and E_2 (610 ms). However, when we have a stream of 1000 frames, the best completion time is the third case (i.e., Multiple TEEs). The reason is *pipelined execution* which means that

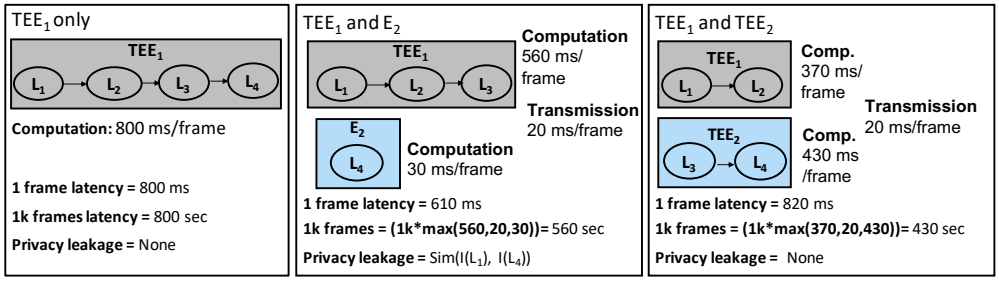

Figure 7.5: Comparison between the execution time of a NN in three different cases: all layers in TEE_1 (left), layers partitioned across TEE_1 and E_2 (middle), layers partitioned across TEE_1 and TEE_2 (right).

when both TEE_1 and TEE_2 are concurrently processing different frames, the completion time becomes bounded by the completion time of the slowest device. In the case of one enclave and one regular CPU (middle column), the enclave has to process more than half of the layers to reach a point where the resolution is less than threshold δ, which makes the bulk of the workload skewed towards the slower device (TEE_1). However, in case of multiple TEEs, the layers can be more evenly divided across the $TEEs$ which results in better chunk completion time than the other two cases. Using multiple enclaves has an additional benefit that it has no privacy leakage because the intermediate data can only be decrypted in TEE_2 unlike the middle case in which the intermediate data can be processed in untrusted processor E_2.

Cost Calculation for different placements: To better illustrate how the completion time of a partitioned NN is calculated, we show an example of the pipelined execution in Figure 7.6. Figure 7.6 depicts the execution for a *chunk* of three video frames. From Figure 7.6, we notice that at the same time that video frame 2 is being processed at TEE_2, video frame 1 is being transmitted over the network and processed at E_2. Hence, the completion time of executing a chunk of n frames on a neural network that follows placement $P_j = (TEE_1, TEE_1, TEE_1, E_2)$ is calculated as:

$$t_{chunk}(n=3,P_j) = 3*(e_{1,TEE_1} + e_{2,TEE_1} + e_{3,TEE_1}) + tr(E_1 \xrightarrow{D_{L_1}} E_2) + e_{4,E_2} \tag{7.4}$$

We note that for a large chunk of frames n, the chunk completion time for the example in Figure 7.6 becomes bounded by the first term only, which is the queuing time in the slowest device TEE_1.

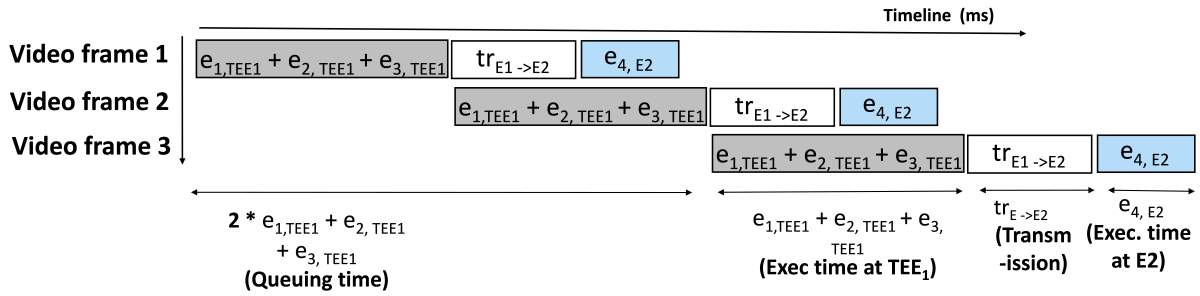

Figure 7.6: Pipelined execution of 3 video frames ($n = 3$) for the partitioned NN in the middle column of Figure 7.5.

$$\begin{aligned} t_{chunk}(n,P_j) &= n*(e_{1,TEE_1}+e_{2,TEE_1}+e_{3,TEE_1})+tr(E_1 \xrightarrow{D_{L_1}} E_2)+e_{4,E_2} \\ &\simeq n*(e_{1,TEE_1}+e_{2,TEE_1}+e_{3,TEE_1}) \end{aligned} \tag{7.5}$$

Algorithm Steps: The following steps are performed offline once to find the best initial placement, then, the system keeps monitoring the online profiling information for the execution time of each NN layer and issues a re-partitioning when the profiling information deviates from the predicted execution times.

1. **Step 1 - Construct placement tree to explore possible partitioning points**: Figure 7.7 shows an example placement tree for a NN with 3 layers. Each node in the tree shows the placement of one or several layers, where $L_1@TEE_1$ denotes that L_1 is placed on TEE_1 while $L_1 \rightarrow L_3@TEE_1$ denotes that L_1, L_2, L_3 are placed on TEE_1. Each path P_j in the placement tree is a possible solution for the privacy-aware placement problem. The first level starts with possible placements in TEE_1 because the processing has to start in a trusted resource. The second level shows possible placements of the second part of the NN on either E_1, E_2, or TEE_2 and the third level of the tree shows the possibility of offloading layers from TEE_2 to E_2. Another level can be added to the tree if the application requires that the final layers get executed where the processing started in TEE_1. This is beneficial if the application requires the object labels to be generated in a trusted resource.

2. **Step 2 - Evaluate the execution time for each path in the placement tree**: For each path P_j in the placement tree, we compute two values: (1) the completion time $t_{chunk}(n,P_j)$ according to Equation 7.5, (2) the maximum similarity computed as:

$$Sim_{P_j} = max(Sim(I(L_1),I(L_x)),\forall p_x \in P_j \; and \; p_x \in V_{R_{UT}} \tag{7.6}$$

This value defines the maximum privacy leakage across the placement path.

3. **Step 3 - Choose the optimal placement path**: From step 3, we get two sets $S_{completion}$ and S_{Sim}. The first set contains completion times and second set contains the similarity (i.e., privacy) value for each path P_j out of the N possible paths in Figure 7.7:

$$S_{completion} = \{t_{chunk}(n, P_j)\}_{j=0}^{j=N} \tag{7.7}$$

$$S_{Sim} = \{Sim_{P_j}\}_{j=0}^{j=N} \tag{7.8}$$

We choose the optimal placement P_j that yields the minimum completion time while the similarity is less than threshold δ:

$$j^* = \underset{j}{\operatorname{argmin}}(S_{completion}) \quad such\ that\ Sim_{P_j} < \delta \tag{7.9}$$

Algorithm analysis: The algorithm complexity is bounded by the number of paths N in the placement tree. As shown in Figure 7.7, the tree has 3 levels. To calculate the complexity, we compute the maximum degree at each level of the tree. We denote the maximum degree at level 1 as deg_1. The maximum degrees at levels 2 and 3 are denoted as deg_2 and deg_3, respectively. The upper bound on the total number of paths $N = O(deg_1 * deg_2 * deg_3)$. Level 1 has M nodes because there are M possible ways to partition a NN of M layers across TEE_1 and another device, therefore $deg_1 = M$. Each node in level 2 denotes where the second part of the NN is executed. The second part can be executed in E_1, E_2, or distributed among TEE_2 and E_2. Since there are $M-1$ ways to distribute $M-1$ layers across TEE_2 and E_2, then $deg_2 = 2 + (M-1) = M+1$. Level 3 will only have one possibility which is deploying the rest of the layers on E_2 so $deg_3 = 1$. Based on the previous analysis, $N = O(M * (M+1) * 1) = O(M^2)$. We note that this analysis is based on the assumption that we have 2 TEEs, in the general case with R TEEs, $N = O(M^R)$, where R is expected to be a small constant significantly lower than the number of layers $R << M$.

7.6 EVALUATION

System setup: The computing infrastructure consists of two desktops. Both desktops have Intel Core i7-9700k (4.6 GHz) CPU and 32 GB of memory. Both desktops are equipped with Intel SGX trusted enclave technology. Each server has an Nvidia RTX 2080 GPU. We control the bandwidth between the two machines to be 30 Mbps which simulates an average wide area

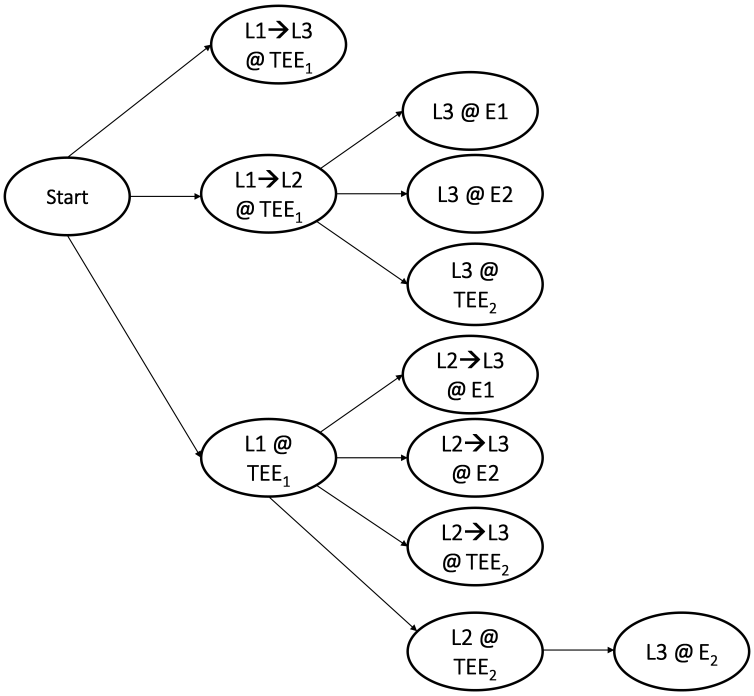

Figure 7.7: Placement Tree that shows possible placements for the example in Figure 7.6. Each path is a solution to the privacy-aware placement problem

network connection. We use Google Asylo [106] to deploy gRPC services in the enclave and the NN inference is done through TFtrusted [107], which provides integration between Asylo, gRPC and TensorflowLite. Each of the devices has a local dataflow engine, Apache NiFi, that handles data transfer between operators within the device and we use Echo [39] orchestration framework to handle the communication between the two Nifi instances.

Models: We evaluate our approach with well-known Convolutional neural network (CNN) models: (1) GoogLeNet, (2) Alexnet, (3) Resnet, (4) Mobilenet, (5) Squeezenet. We download the models from the Tensorflow Model Library [108], pre-trained with the Imagenet (ILSVRC 2012) dataset.

Datasets: We experiment with three video surveillance datasets [82]. The datasets vary in the object types (car, person, boat), and locations (indoor, outdoor, different cities). We experiment with surveillance datasets because they include sensitive content such as faces and car license plates that require processing in a privacy-preserving fashion. Our dataset consists of 1 hour from each surveillance video and we sample one frame per second so we get a total of 10800 frames. The resolution of each frame is restricted to 224x224 pixels because it is the input resolution required by all the models.

7.6.1 Latency and Resolution of Intermediate Layers Output

In this section, we conduct an experiment to show the relationship between the latency of computing the output of intermediate NN layers and the similarity between the intermediate output compared to the original image. For each of the five NN models, we pass an individual video frame to the NN and we compute the latency of computing each of the intermediate layer outputs. To visualize the intermediate layer, we convert the tensor to a grid of small images (Figure 7.4) using TensorFlow CNN visualization tool. To assess the similarity between intermediate layers and the original image, we get the resolution of a single image in the grid. The resolution of a single image gives an estimate of the privacy leakage from this intermediate layer. In Figure 7.8, we plot the relationship between the percentage of time spent in computing the intermediate output vs. the resolution of this intermediate output. Figure 7.8 shows that the deeper the layer is, the more time one spends to compute its output and the less its resolution (i.e., correlation with the original input) will be. However, an interesting insight in Figure 7.8 is that different models tend to have different trends. For example, for models like GoogLeNet, Squeezenet, one needs to spend 80% of the entire inference time to reach an intermediate output with resolution of 20x20 pixels or less. However, Alexnet and Resnet reach such resolution in less than 50% of the inference time. From this experiment, we conclude that each model can be partitioned differently based on how fast the resolution drops. We will discuss different partitioning strategies in Section 7.6.3.

7.6.2 User Study to find Resolution Threshold

In this section, we conduct user study to show the relationship between the resolution of the intermediate layer's output and the ability of human subjects to recognize the objects in the image. The study is divided into two parts. In the first part, we display the intermediate output of several layers from each model and we ask each user to identify the object in this intermediate output. In the second part of the survey, we show the users an original image and 5 sample images, where each image is an output of a random layer of the same NN model. The users are asked to rank these images based on how representative they are compared to the original image. Ten subjects took part in this study. They had normal/corrected-to-normal vision.

Data Generation Process: For the first part of the study, we collect 100 images from Imagenet's image classification dataset. Each image has a single object and the objects belong to 10 classes: Cat, Dog, Car, Truck, Bus, Aeroplane, Boat, Horse, Elephant, and Person. We use the same 5 NN models, described in the previous section, to generate the intermediate output. For each model, we pick 5 layers that have distinct resolution values. We choose a random image from the dataset to generate the output of each layer. We end up with 25 outputs for this part of the survey and we

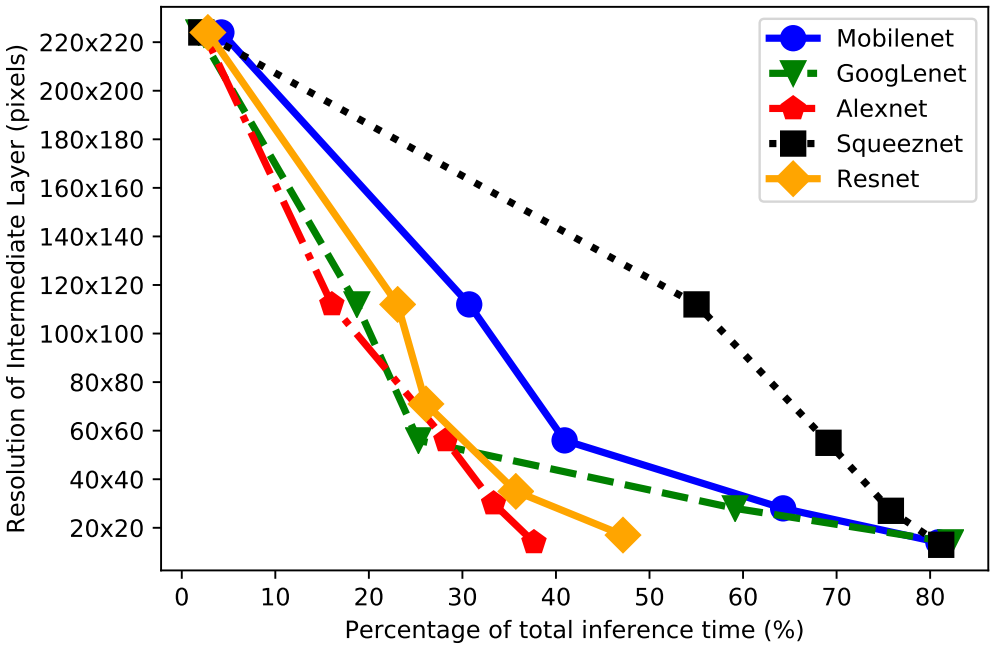

Figure 7.8: Relationship between the percentage of time spent in executing inference and the resolution of the intermediate output.

ask the users to identify the object in each of the 25 images. For the second part of the survey, we choose one image at random for each model, then we pass the image by different layers of the model, and we generate the output of 5 layers that have distinct resolution values. We shuffle the order of the layers' outputs and we ask the users to rank them based on the similarity to the original image. Figure 7.9 shows an example question shown to the users.

Protocol: Each subject is asked to fill the survey individually. We present each subject with a link to a web form with 30 questions: in 25 questions he/she is asked to identify the object in the image and in 5 of them he/she was asked to sort images based on their similarity to the original image. We ask the subjects to resize the images as much as they can to try to identify the objects.

Discussion and Results: In the first part of the survey, we calculate the accuracy at which the users were able to identify the objects. In Figure 7.10, we plot the accuracy with respect to the resolution values. The figure shows that human subjects were able to identify the object with 100% accuracy when the resolution is above 110x110. The accuracy degrades slightly when the resolution is in the range 26x26 - 32x32 pixels, and it drops drastically when the resolution is in the range 12x12 - 18x18 pixels. Hence, we conclude that a resolution below 20x20 pixels is the sweet spot at which the objects in an image are hardly identifiable. For the second part of the survey, we evaluate how often the ranking of humans matches the ranking based on the highest to lowest resolution. For each question (e.g., Figure 7.9), we rank the images from 1 to 5 based

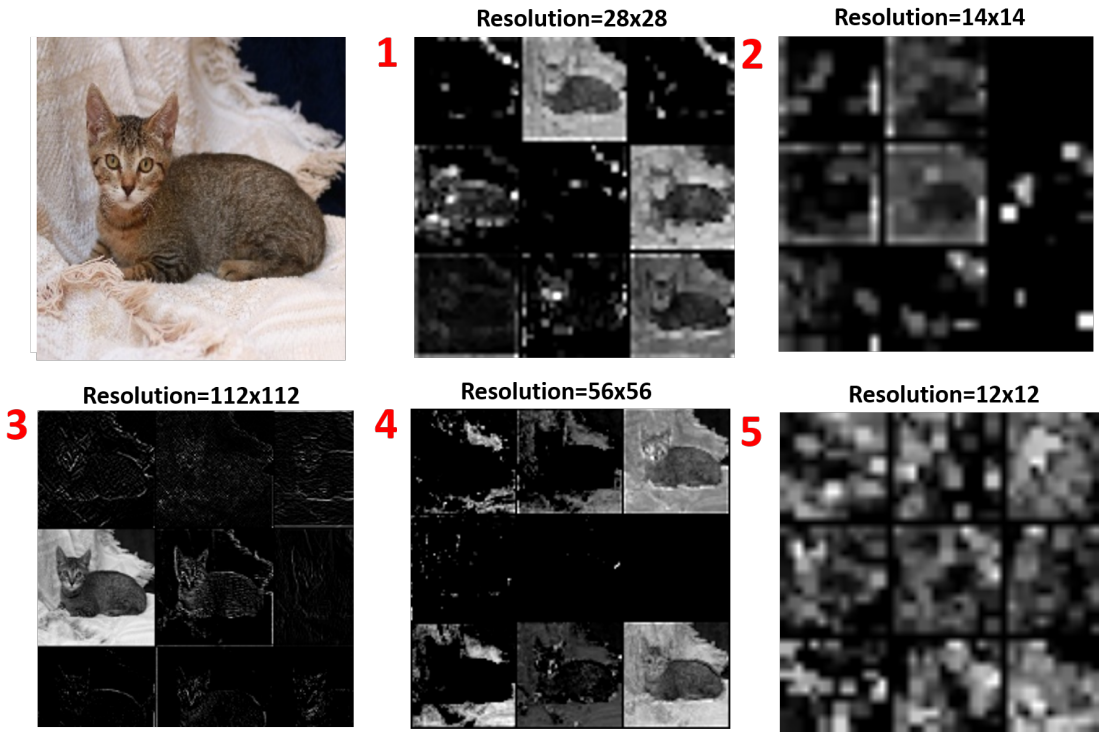

Figure 7.9: Example question in the second part of the user survey. Users are asked to rank the images according to the similarity to the original image (Resolution values are not shown to users).

on the resolution where rank 1 represents the highest resolution and rank 5 represents the lowest resolution. For each ranked image, we calculate the percentage of human subjects who ranked it the same way as the resolution. We plot the results in Figure 7.11. The results show an interesting insight that human subjects have different opinions about which images rank as the most similar to the original image (rank 1). On the other hand, there was a general consensus among human subjects and resolution metric about which images should rank last (ranks 4 and 5). We conclude that when the resolution is high, the objects are well identifiable and human subjects rank images differently, but everybody agrees on the ranking when the image resolution is below 20x20 pixels.

Finally, since resolution cannot be solely used as a metric for privacy, we define the privacy leakage as similarity between original image vs intermediate image and the ability of human subjects to recognize the concepts. We measure this metric through the Pearson correlation coefficient between the original and intermediate image. The more correlation exists, the more similar the two images are. The output of a layer is considered private if the pearson correlation similarity between its output and the input image is below a certain threshold. We decide to use pearson correlation because unlike mean square error (MSE) and peak signal-to-noise ratio (PSNR), it does not rely on the difference in *absolute value* (i.e., color) between corresponding pixels but it measures the correlation between the *structure* of the two images. In Figure 7.12, we plot the accuracy

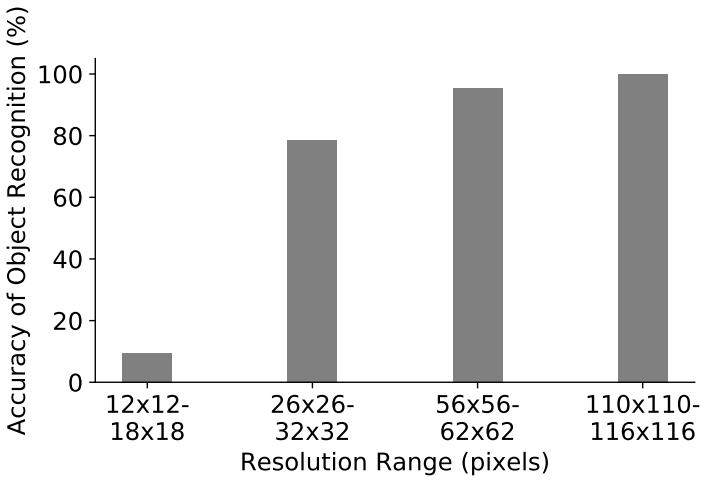

Figure 7.10: Accuracy of object recognition by human subjects at different resolution ranges (Lower accuracy signifies better privacy)).

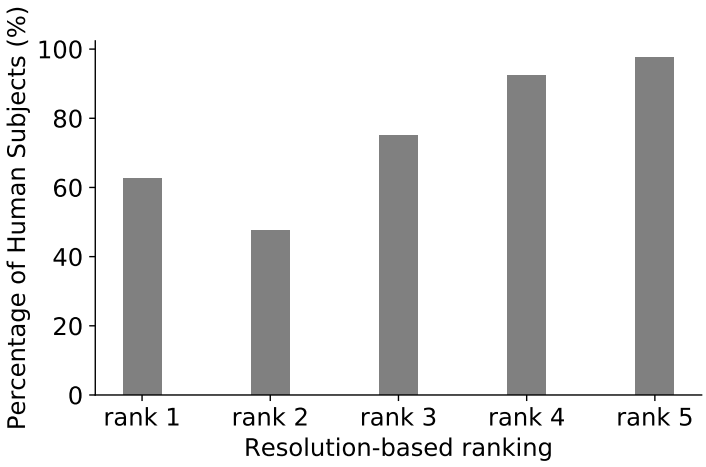

Figure 7.11: Percentage of human subjects who rank an image in the same position as resolution-based ranking.

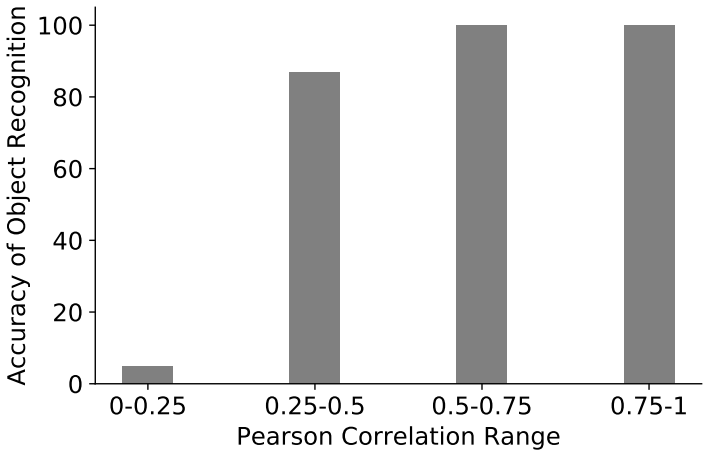

Figure 7.12: Accuracy of object recognition by human subjects at different pearson correlation ranges. (Lower accuracy signifies better privacy and shows that pearson correlation can be used as a metric for privacy)).

with respect to the Pearson correlation values. The figure shows that the accuracy at which human subjects can identify objects degrade significantly when the pearson correlation is less than 0.25. Hence, we conclude that a correlation below 0.25 has the least privacy leakage. We note that our neural network partitioning method that we evaluate in the next section is not restricted to using Pearson correlation as a metric and more complex similarity metrics can be utilized.

7.6.3 Neural Network Partitioning

In this section, we show the impact of our neural network partitioning. We compare between the following strategies:

1. **1 TEE:** the entire NN is deployed in one TEE,

2. **No pipelining:** this approach is adopted by related work in mobile computing literature [68] and a recent NN inference system (Neurosurgeon [59]). In those approaches, NN partitioning is designed to minimize the latency of one frame ($n = 1$) and they do not consider the cases when a stream of frames arrives in a short period of time.

3. **1 TEE & 1 GPU:** We use the proposed method in Section 7.5 to find the layer to offload to the GPU such that the layer we partition at has an output with resolution less than 20x20 pixels or correlation less than 0.25 (In this approach, we do not consider having the second TEE in the available resources).

4. **2 TEEs:** The neural network is partitioned across two TEEs according to the algorithm in Section 7.5 (In this approach we do not consider offloading to a CPU/GPU).

5. **Proposed:** We partition the neural network considering all available resources (2 TEEs and one GPU).

Figure 7.13 shows the speedup in the end-to-end execution of the entire dataset of 10800 frames for the different approaches compared to the baseline that uses 1 TEE for the entire processing. For the approaches that employ partitioned NN, the speed up is measured on the end-to-end execution time which includes: the time to process both parts of the NN, and the time to encrypt and transmit intermediate outputs. From the results, we observe that for three of the above models (GoogleNet, Mobilenet and Squeezenet) using 2 TEEs outperforms using 1 TEE and 1 GPU because in the latter case most of the processing happen in the slow TEE which results in 1.15-1.5x speedup over using 1 TEE for the entire execution. On the other hand, 2 TEEs can achieve 1.8 to 1.95x speedup for the same three models because the model processing is almost equally distributed across the enclaves. However, the results are different in the other 2 models (Alexnet, Resnet) because the majority of the workload happens in the fast GPU resulting in 2.5-3.1x for 1 TEE & 1 GPU compared to a speedup of 2.2-2.3x for 2 TEEs.

We note that the proposed approach can go beyond using 2 TEEs by using 2 TEEs and 1 GPU resulting in the best speed up 3.2-4.7x. The best speedup happens in Alexnet because each TEE can do only 19% of the processing leaving the remaining 62% to the fast GPU. Our final observation on Figure 7.13 is that the *No pipelining* baseline ends up choosing the same decision as 1 TEE & 1 GPU because its partitioning decision is based on one frame only and it fails to consider that the second TEE used to process the next frame while the first TEE is processing the current frame.

7.6.4 Inference Latency and Overhead on TEE

In this section, we analyze the time it takes to perform an inference on a single frame using one vs. two TEEs. We note that in the case of two TEEs we analyze the time to process the first part of the NN in TEE_1, the time to encrypt the intermediate output in TEE_1, the time to transmit the intermediate output to TEE_2, the time to decrypt the intermediate output, and, the time to process the second part of the NN in TEE_2. We note that not all these times are added together in the presence of a stream of frames because of pipeline parallelism but in this section we show it for a single frame. We show a breakdown of the different execution times in Figure 7.14. The figure shows that distributing the layers across the enclaves have an additional benefit that it reduces the memory requirements per enclave resulting in a faster overall execution. As shown in the figure the sum of execution times in each enclave is less than the execution time of the entire network in

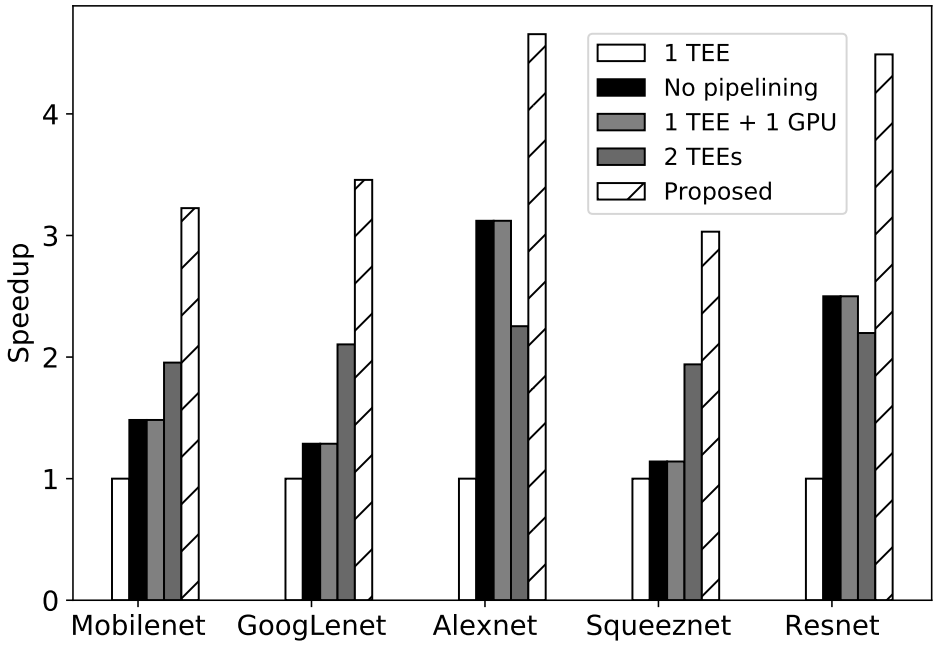

Figure 7.13: Speedup comparison between various NN partitioning approaches.

one enclave in 4 of the 5 models. The result is more pronounced in Alexnet because it is the largest model (243 MB). Conversely, Squeezenet does not show a similar trend because it is the smallest model (5 MB) and partitioning its layers does not have a significant impact on the memory. We notice that the encryption and decryption times using Advanced Encryption Standard (AES) with 128-bit key is negligible (i.e., less than 2.5 ms/frame) compared to the other times so we omit it from the figure. The transmission time ranges from 0.01 to 0.12 seconds based on the size of the intermediate layer and it is less significant than computation time. The computation time represents the largest portion of the processing due to the limited hardware resources in the TEE, the computation time ranges from 1.1 seconds for Squeezenet to 7.2 seconds for Resnet.

7.7 DISCUSSION

7.7.1 Other Deep Learning Models:

The deep learning models we experimented with in this thesis belong to the object detection/-classification category which represents an important and widely deployed subset of all video processing. However, many of these models produce lower resolution images deeper down the network which is not the case for other deep learning architectures such as UNet architecture [109]

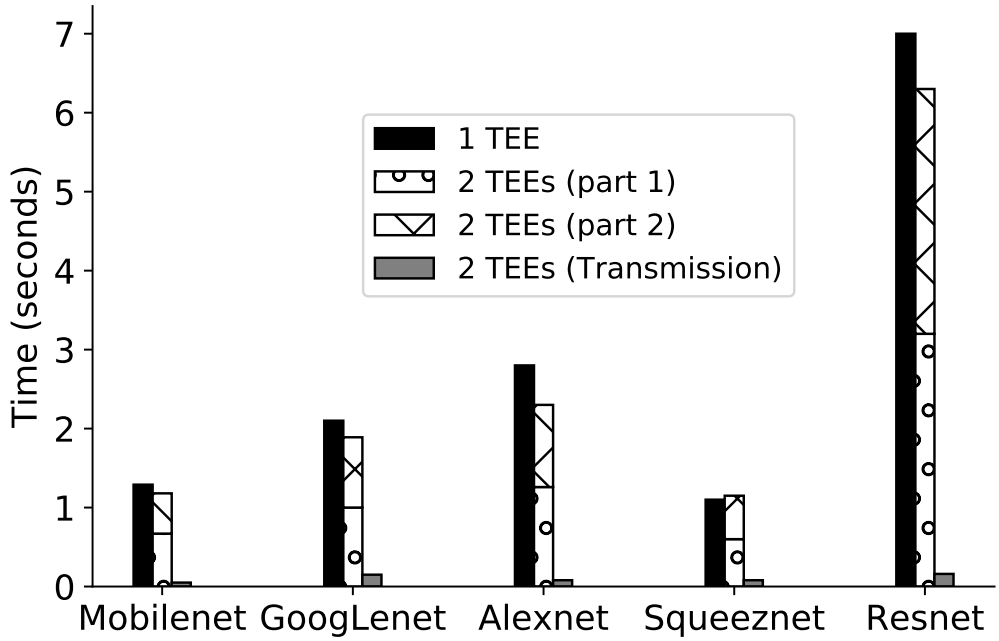

Figure 7.14: Execution time of NN inference per frame when deployed in 1 TEE and 2 TEEs.

which is the state of the art for image segmentation tasks. For UNet-like architectures, our approach should handle it using the multiple enclaves setup in which each subset of the model is executed inside a trusted enclave device. We note that one of the fundamental aspects that make us generalize our approach to other neural network models and other analytics workloads is that we base our framework on four main concepts which most visual IoT applications can be defined by. These four concepts are the application graph, resource graph, data model, and utility functions. Once we define these components, we can apply our approaches to different neural network models, compute resources and data types. The concepts are described in more detail in Section 1.4.

7.7.2 Security Attacks and Guarantees:

Based on the research efforts in understanding the internal mechanisms of DNNs [81], we conclude that for an image classification DNN, the first few layers (shallow layers) represent low-level image processing operations such as edges, corners, and contours of the original inputs. On the other hand, later layers (i.e., deep layers) represent more abstract and class-specific information related to the final outputs. Mapping these insights to a security domain, we conclude that the output of shallow layers discloses different information than that of deeper layers. In the following, we describe the information leakage from shallow and deep layers and how an adversary can

use this information uncover the contents of the original input f_y after obtaining the input to the intermediate layer $I(L_x)_y$ that sits outside of the enclave. We consider that the adversaries A1,A2, and A3 have no prior knowledge of input f_y, but they have different attack strategies: (1) A1 intends to visually analyze the exposed information to infer information about the original input, (2) A2 intends to reconstruct the original inputs from the exposed information, (3) A3 intends to manipulate the system into exposing information about the input to another layer L_z.

Input Inference from output of shallow layers (A1): Shallow layers may still contain low-level information of the original inputs. Hence, it is straightforward for humans, acting as adversaries, to understand the output of shallow layers by projecting them to pixel space as shown in Figure 7.4. We consider that such intermediate outputs at shallow layers explicitly disclose the visual information of the original inputs. By progressing towards deeper layers, the output of the layers in pixel space are not directly comprehensible as illustrated in 7.4 and as detailed in the user study(Figures 7.10, 7.11). In our user study we concluded that an image of resolution 20x20 pixels is hardly identifiable by humans acting as adversaries and they possess very low correlation with the original image (≤ 0.25). However, the information is still preserved within the layers and is crucial for final classification decisions, if adversaries can obtain both the preceding intermediate outputs and layer parameters. Adversaries will then be able to extract sensitive information of the original inputs by exploiting the input reconstruction which will be discussed next.

Input Reconstruction Attacks (A2): Previous research [110] has shown some attacks that intend to reconstruct the original input from the intermediate layers output of a NN. Such attacks are also known as *Input Reconstruction Attacks*. To perform such attacks, the adversary has to either have: (1) the model parameters that lead the input image to be transformed to the intermediate layer output, or (2) the ability to query the model to generate pairs of images and their corresponding intermediate output, such pairs can be used later to recover the model parameters. Our framework prevents this attack because the model parameters can only be decrypted inside the enclave that is rented by the user. The attacker (cloud provider) cannot see the model parameters that are executed inside the enclave. In addition, the user can add an authentication mechanism [62] to block any external adversaries from querying their enclave device which prevents the adversary from obtaining image and intermediate output pairs.

Control plane Attacks (A3): We note that an integral part of the Serdab system is the edge-cloud orchestration framework (Figure 7.2) which provides an abstraction for deployment and management of NN layers across edge 1 and edge 2 dataflow engines. The orchestration engine has several control components that are crucial to ensure the data privacy. For example, the privacy-aware placement service decides the placement that ensures the least privacy leakage. The placement service also relies on the resource manager to provide correct information about the available resources in each device. Hence, if such components are placed on an untrusted device,

an internal administrator acting as an adversary can tweak the privacy-aware placement service to deploy only one layer on the trusted enclave, hence exposing a shallow layer's output to the untrusted device. To mitigate this, we recommend the deployment of the edge-cloud orchestration framework and its sub components on a local device that is strictly controlled and managed by the data owner. We note that the orchestration framework requires minimal compute resources because it is not involved with the actual processing of data and it only needs to send control commands to the dataflow engines on the TEE and GPU devices. Another solution to ensure trust in the orchestration framework is to deploy it as a service inside a TEE device in edge device 1 or edge device 2. This ensures the protection of the orchestration framework from attacks by malicious attackers or misuse by untrusted administrators.

7.8 CONCLUSION

In this chapter, we present Serdab, a framework for analyzing video streams across multiple enclave devices to preserve data privacy. Serdab allows partitioning deep neural network layers across multiple devices. We leverage two insights to find the best placement of NN layers across devices. The first insight is that the intermediate output of shallow NN layers tends to be more similar to the original input than the output of deeper layers, hence the enclave can run only the shallow layers until the output is not correlated to the original input. We validated our findings by conducting a user study and we realize that users cannot figure out the objects in the image when the resolution of layer's output is less than 20x20. The second insight that we leverage in this work is pipeline parallelism which happens when both enclaves are concurrently processing different frames which improves the completion time compared to running the majority of the workload in one enclave. Our results show that for a stream of 10,800 frames, our partitioning strategy achieves up to 4.7x speedup compared to executing the entire neural network in one enclave.

CHAPTER 8: CONCLUSION AND FUTURE WORK

8.1 DISSERTATION SUMMARY

In this dissertation, we discussed several advances and technologies that enable automated analysis of visual IoT data across the edge and the cloud. We argue that the current video analytics systems are not keeping up with such advances and we lack a holistic approach that unfolds and addresses new challenges that arise from: (1) Leveraging new technologies such as Serverless computing, and Privacy-preserving machine learning, (2) Keeping up with recent advances in machine learning, computer vision, and IoT.

We showed that the above challenges can be addressed through careful consideration of application deployment in the presence of such new advances. For new technologies such as Serverless computing and privacy preserving machine learning: (1) we devised models that can reason about the tradeoff between latency, locality, privacy, and pricing and, (2) we developed algorithms that can achieve privacy and pricing guarantees with marginal effect on the latency. Throughout this work, we showed how **Function Fusion** is an effective technique to reduce the price of state transitions in serverless computing platforms. We also showed how **Neural Network Partitioning** across multiple TEEs can effectively reduce the processing latency while preserving the privacy of the data being analyzed.

In addition to addressing challenges associated with new technologies, we addressed the challenges arising from keeping up with recent advances in relatively older technologies such as machine learning, computer vision, and IoT. For example, since advances in computer vision have made it possible for more data to be analyzed by algorithms rather than humans, we devise **Semantic Video Encoding** that makes video compression more aware of the underlying analysis algorithms which is in contrast with the long standing assumption that video compression algorithms has to be designed to please human viewers. Semantic video encoding aims to minimize bandwidth between edge and cloud by allowing the application to analyze key-frames only which constitute less than 4% of the video.

Advances in IoT devices and the vast amount of data they can produce in split seconds has also led us to explore **Function Placement** in a different perspective. Our approach to function placement relies on **pipeline parallelism** in which we consider that both the edge device and the cloud server are concurrently executing different video frames. This pipelining happens because the interarrival time between frames is less than the processing time of one frame which allows both the edge and the cloud to be concurrently processing different frames.

8.2 LESSONS LEARNED

Through the development of four optimization algorithms for visual IoT analytics systems, we learn some crucial guidelines that we summarize in Table 8.1. The first step is to identify the main utilities that one is interested in optimizing. We learned through this dissertation that different companies/organizations care about different utilities. For example, healthcare and financial institutions care about privacy more than other utilities due to the sensitivity of the data they store. Self-driving car companies care about latency and reliability because they need to take decisions in split-seconds. On the other hand, agriculture/manufacturing companies care about bandwidth because many of their devices are located in rural areas with limited connectivity to broadband networks. The price is also a common utility whenever any of such companies utilize public edge/cloud infrastructure. After identifying the main utility, the second and third steps are to clearly define the application model, and underlying compute resources.The application model gives guidance about whether the operations are memory, compute or I/O intensive which influences the design of the algorithm. The available resources also help to identify the main hardware limitations such as the 128 MB memory limitation in TEEs. After identifying the application and resource models, the next step is to build a prototype of the application to identify the main bottlenecks, tradeoffs, and the crucial insights that help us optimize the system functionality. For example, in the Costless system, a key insight that we realized is that most of the price is not paid for the function execution but is rather paid for transition from one function to another; Another insight that we realized in the Droplet system is that the interarrival time between frames is much faster than the time to process a single frame which presents an opportunity for pipeline parallelism. After identifying the insights and bottlenecks, the final step is to leverage such insights to develop an algorithm that optimizes the main utility of interest; For example, we leveraged Costless insights to develop an algorithm that explores fusing functions to reduce transition price. Similarly, we leveraged the insights that TEEs cannot load an entire NN model in their constrained-memory and we developed an algorithm to distribute NN layers across multiple TEEs. Table 8.1 summarizes the rest of the insights and algorithms.

8.3 FUTURE DIRECTIONS

8.3.1 On-device Machine Learning (Federated Learning)

In this dissertation, we focus on the analysis operations that are performed near-real time. This description applies to ML inference from pre-trained models rather than ML training. Hence, so far we assumed that ML models are trained in the cloud and then deployed across the edge

	Bandwidth Optimization	**End-to-End Latency Optimization**	**Price Optimization**	**Privacy preserving deployment**
Main Utility	Bandwidth	Latency	Price	Privacy
Application Model	Compression /Decompression + one NN model	Arbitrary number of high-level or low-level operators	Arbitrary number of Lambda Functions (high-level operators)	A NN model with arbitrary number of layers
Underlying Resources	Camera, Private Edge, Private Cloud	Private Edge, Private Cloud	Public Edge, Public Cloud (Serverless Computing)	Public Edge, Public Cloud (Includes TEEs)
Bottlenecks /Insights	95% of the frames are decompressed then thrown away because they are irrelevant	Interarrival time between frames is less than processing time of a single frame	(1) Edge price is fixed no matter how many functions are executed (2) Cloud price is not just per function execution but also for transition from one function to another	(1) TEEs have memory constraints when they execute an entire NN (2) objects are non-identifiable in the intermediate output of deeper NN layers
Tradeoffs	Bandwidth, Latency, Accuracy	Latency, Locality	Latency, Locality, Price	Latency, Locality, Privacy
Approach	Semantic Encoding	Pipeline parallelism / Operator placement	Function Fusion / Function placement	NN partitioning across TEEs

Table 8.1: Guidelines for developing optimization algorithms for visual IoT analytics systems

and cloud. However, for privacy reasons, a new paradigm has emerged to allow ML training to be performed locally on users' edge devices (e.g., smartphones). This allows users to train personalized models on their devices while preventing their sensitive data from leaving the user's device. The new paradigm is known as **Federated Learning**, the training procedure of federated learning goes in multiple rounds. In each round, the training is performed in four steps: First, each user device updates its local model with the data captured/stored locally in the device. Second, each user device regularly sends its updated model to a centralized server. Third, the centralized server aggregates updated models from various devices to form a global model, and finally the global model is sent back to the all users' devices to start a new round of training. The main disadvantage of federated learning is the significant energy and computation cost incurred by the battery-operated user devices (smartphones). To address this problem, we explore an alternative design for *Federated learning*, which we refer to as *Trusted Federated Learning (TFL)*. In TFL, rather than performing the computationally intensive training on the user's smartphone, we allow multiple users to send their private data to a Trusted execution environment (TEE) to perform the training on their behalf. The TEE provides the same privacy guarantees on the data privacy and it has three added advantages: (1) it relieves the energy-constrained smartphones from performing the training computation. (2) One TEE can aggregate private data from multiple smartphones which makes the model training/convergence significantly faster. (3) The data sent from the device to the TEE is compressed images/videos which is much more bandwidth-efficient compared to floating-point tensors. The proposed design to Federated learning makes an interesting extension to this dissertation because it extends the dissertation to support ML training scenarios on edge devices in addition to the ML inference scenarios that are already studied through the course of the dissertation.

8.3.2 ML Inference on Video Clips

ML inference on videos consists of passing either individual video frames or a group of video frames (i.e., clip) to a pre-trained ML model. In this dissertation we have seen many applications that require running ML inference once per frame such as object detection. However, there exist other applications such as activity recognition in which a group of frames (e.g., 60-120 frames) is needed to recognize the activity being performed in the video (e.g., walking, running, jumping ,.etc); State-of-the-art deep learning-based approaches for activity recognition are based on a two-stream convolutional neural network (CNN) [111][112] architecture; In two-stream architectures, videos can naturally be decomposed into spatial and temporal components. The spatial part provides information about scenes and objects of the video, taking a single frame as input. Nevertheless, the temporal part, which consists of stacked optical flow vectors, shows the movement of

the objects in the form of motion across the frames. This way, the authors divide the architecture into two streams. Each stream is implemented using a CNN and the scores from both streams are combined using a support vector machine (SVM). In this dissertation, we have addressed a single-stream CNN for object detection/recognition, an interesting future direction is to explore how analyzing a group of frames on a two-stream architecture will affect the latency, bandwidth, and real-time nature of the application. We are also interested in exploring how such changes will impact our placement of computation across the edge and cloud.

REFERENCES

[1] *IHS video surveillance trends*, https://technology.ihs.com/532501/, Last accessed December 2018.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, Oct 2016.

[3] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[7] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein et al., "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[8] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[9] K.-H. Yu, C. Zhang, G. J. Berry, R. B. Altman, C. Ré, D. L. Rubin, and M. Snyder, "Predicting non-small cell lung cancer prognosis by fully automated microscopic pathology image features," *Nature communications*, vol. 7, p. 12474, 2016.

[10] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 9, pp. 1627–1645, 2009.

[11] B. Heisele, P. Ho, and T. Poggio, "Face recognition with support vector machines: Global versus component-based approach," in *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, vol. 2. IEEE, 2001, pp. 688–694.

[12] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld, "Face recognition: A literature survey," *ACM computing surveys (CSUR)*, vol. 35, no. 4, pp. 399–458, 2003.

[13] *Bandwidth Requirements for Nest Cameras*, https://nest.com/support/article/How-much-bandwidth-will-Nest-Cam-use#nest-camera-bandwidth-upload-iq, Last accessed February 2019.

[14] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE PerCom*, vol. 8, no. 4, 2009. [Online]. Available: http://dx.doi.org/10.1109/MPRV.2009.82

[15] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: Optimizing neural network queries over video at scale," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1586–1597, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137664

[16] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, V. Bahl, M. Philipose, P. B. G. , and O. Mutlu, "Focus: Querying large video datasets with low latency and low cost," *USENIX OSDI*, October 2018.

[17] G. Adzic and R. Chatley, "Serverless computing: Economic and architectural impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3106237.3117767 pp. 884–889.

[18] *Amazon AWS Lambda*, https://aws.amazon.com/lambda/.

[19] *Amazon AWS Greengrass*, https://aws.amazon.com/greengrass/.

[20] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *PMLR 2016*.

[21] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *CCS '17*.

[22] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "Deepcache: Principled cache for mobile deep vision," in *MobiCom '18*.

[23] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *MobiSys '17*.

[24] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," *CoRR*, vol. abs/1803.05961, 2018.

[25] N. Hynes, R. Cheng, and D. Song, "Efficient deep learning on multi-source private data," *CoRR*, vol. abs/1807.06689, 2018. [Online]. Available: http://arxiv.org/abs/1807.06689

[26] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Report 2016/086, 2016, https://eprint.iacr.org/2016/086.

[27] *Intel Software Guard Extensions Remote Attestation End-toEnd Example.*, https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example., Last accessed November 2019.

[28] T. Elgamal, A. Sandur, P. Nguyen, K. Nahrstedt, and G. Agha, "DROPLET : Distributed operator placement and resource provisioning for iot applications spanning edge and cloud resources," in *IEEE CLOUD*, 2018.

[29] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni et al., "Storm@twitter," in *ACM SIGMOD'14*.

[30] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737 pp. 423–438.

[31] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink^{TM}: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, pp. 28–38, 2015.

[32] *Apache Nifi*, https://nifi.apache.org/docs/nifi-docs/html/overview.html.

[33] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, Nov 2015, pp. 73–78.

[34] *Apache Edgent, v1.1.0*, http://edgent.apache.org/.

[35] *IBM Node-RED*, https://nodered.org/.

[36] *Apache MiNiFi*, https://nifi.apache.org/minifi/index.html.

[37] *Eclipse Kura*, http://www.eclipse.org/kura/.

[38] *VMware Liota: Little IoT Agent*, https://github.com/vmware/ liota.

[39] P. Ravindra, A. Khochare, S. P. Reddy, S. Sharma, P. Varshney, and Y. Simmhan, "Echo: An adaptive orchestration platform for hybrid dataflows across cloud and edge," *International Conference on Service-Oriented Computing (ICSOC)*, 07 2017.

[40] *Azure IoT Edge*, https://azure.microsoft.com/en-in/ campaigns/iot-edge/.

[41] *Amazon AWS Step Functions*, https://aws.amazon.com/step-functions/.

[42] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: Scalable adaptation of video analytics," *ACM SIGCOMM*, August 2018. [Online]. Available: https://www.microsoft.com/en-us/research/publication/chameleon-video-analytics-scale-via-adaptive-configurations-cross-camera-correlations/

[43] D. Kang, P. Bailis, and M. Zaharia, "Blazeit: An optimizing query engine for video at scale."

[44] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[45] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.

[46] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "Awstream: Adaptive wide-area streaming analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3230543.3230554 pp. 236–252.

[47] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al., "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[48] *Intel® Neural Compute Stick*, https://software.intel.com/en-us/movidius-ncs.

[49] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of fpga-based neural network accelerator," *arXiv preprint arXiv:1712.08934*, 2017.

[50] S. Chen, A. Jain, Z. Gao, K. Nahrstedt, A. Arefin, and R. Rivas, "Metadata-based activity analysis in 3d tele-immersion," in *2016 IEEE Second International Conference on Multimedia Big Data (BigMM)*. IEEE, 2016, pp. 346–353.

[51] L. Galteri, M. Bertini, L. Seidenari, and A. Bimbo, "Video compression for object detection algorithms," 08 2018, pp. 3007–3012.

[52] C. Chen, J. Cai, W. Lin, and G. Shi, "Surveillance video coding via low-rank and sparse decomposition," 10 2012, pp. 713–716.

[53] X. Guo, S. Li, and X. Cao, "Motion matters: A novel framework for compressing surveillance videos," 10 2013, pp. 549–552.

[54] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94

[55] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw pp. 613–627.

[56] *TensorFlow serving*, 2016, https://www.tensorflow.org/serving.

[57] W. Wang, S. Wang, J. Gao, M. Zhang, G. Chen, T. K. Ng, and B. C. Ooi, "Rafiki: Machine learning as an analytics service system," *CoRR*, vol. abs/1804.06087, 2018.

[58] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, V. Bahl, and M. Philipose, "Videoedge: Processing camera streams using hierarchical clusters," *ACM/IEEE Symposium on Edge Computing (SEC)*, 2018.

[59] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 2, pp. 615–629, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3093315.3037698

[60] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," *CoRR*, vol. abs/1806.03287, 2018.

[61] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song, "Occlumency: Privacy-preserving remote deep-learning inference using sgx."

[62] Z. Gu, H. Huang, J. Zhang, D. Su, A. Lamba, D. Pendarakis, and I. Molloy, "Securing input data of deep learning inference systems via partitioned enclave execution," *CoRR*, vol. abs/1807.00969, 2018. [Online]. Available: http://arxiv.org/abs/1807.00969

[63] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *ACM DEBS*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2933267.2933312 pp. 69–80.

[64] X. Gu and K. Nahrstedt, "Distributed multimedia service composition with statistical qos assurances," *IEEE Transactions on Multimedia*, vol. 8, no. 1, pp. 141–151, Feb 2006.

[65] L. Ying, Z. Liu, D. Towsley, and C. H. Xia, "Distributed operator placement and data caching in large-scale sensor networks," in *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, April 2008.

[66] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *ACM DEBS*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2488222.2488267

[67] Y. H. Kao, B. Krishnamachari, M. R. Ra, and F. Bai, "Hermes: Latency optimal task assignment for resource-constrained mobile computing," *IEEE TMC*, vol. 16, no. 11, 2017.

[68] L. Yang, J. Cao, H. Cheng, and Y. Ji, "Multi-user computation partitioning for latency sensitive mobile cloud applications," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2253–2266, Aug 2015.

[69] R. Ghosh and Y. Simmhan, "Distributed Scheduling of Event Analytics across Edge and Cloud," *ArXiv e-prints*, Aug. 2016.

[70] T. Korkmaz and M. Krunz, "Multi-constrained optimal path selection," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 2, 2001, pp. 834–843 vol.2.

[71] M. Armbrust, R. S. Xin, C. Lian et al., "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD*. ACM, 2015, pp. 1383–1394.

[72] T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen, "Spoof: Sum-product optimization and operator fusion for large-scale machine learning." 2017.

[73] *TensorFlow XLA (Accelerated Linear Algebra)*, https://tensorflow.org/performance/xla/.

[74] J. Dejun, G. Pierre, and C.-H. Chi, "Ec2 performance analysis for resource provisioning of service-oriented applications," in *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, 2010.

[75] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Grid Computing (GRID), 2010*. IEEE, 2010, pp. 41–48.

[76] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *SC' 2011*, 2011, p. 49.

[77] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *ICDCS*, June 2011, pp. 559–570.

[78] S. Chaisiri, B. S. Lee, and D. Niyato, "Optimization of resource provisioning cost in cloud computing," *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 164–177, April 2012.

[79] M. R. Rahman, "Risk aware resource allocation for clouds," Tech. Rep., 2011.

[80] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing amazon ec2 spot instance pricing," *ACM Transactions on Economics and Computation*, vol. 1, no. 3, p. 16, 2013.

[81] J. Yosinski, J. Clune, A. M. Nguyen, T. J. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," *CoRR*, vol. abs/1506.06579, 2015. [Online]. Available: http://arxiv.org/abs/1506.06579

[82] *TensorFlow serving*, 2017, https://github.com/stanford-futuredata/noscope.

[83] *Taipei Surveillance Camera*, https://www.youtube.com/watch?v=62vCq0s_a88, Last accessed March 2019.

[84] *Amsterdam PTZ Camera*, https://www.youtube.com/watch?v=R5iR3aPryM8&feature=youtu.be, Last accessed March 2019.

[85] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM 2016*, 2016.

[86] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618

[87] A. Sandur, T. Elgamal, G. Agha, and K. Nahrstedt, "Complexity analysis of distributed operator placement for IoT applications," http://hdl.handle.net/2142/99062, 2018.

[88] M. R. Garey and D. S. Johnson, "Complexity results for multiprocessor scheduling under resource constraints," J. A. Stankovic and K. Ramamritham, Eds. IEEE Computer Society Press, 1989. [Online]. Available: http://dl.acm.org/citation.cfm?id=76933.76948

[89] P. G. Bradford, G. J. E. Rawlins, and G. E. Shannon, "Efficient matrix chain ordering in polylog time," *SIAM Journal on Computing*, 1998. [Online]. Available: https://doi.org/10.1137/S0097539794270698

[90] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962. [Online]. Available: http://doi.acm.org/10.1145/368996.369025

[91] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, July 1987. [Online]. Available: http://doi.acm.org/10.1145/28869.28874

[92] T. Elgamal, B. Chen, and K. Nahrstedt, "Teleconsultant: Communication and analysis of wearable videos in emergency medical environment," in *ACM MM'17*.

[93] *Facial Paralysis Institute Photo Gallery*, https://www.facialparalysisinstitute.com/photo-gallery/.

[94] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *FGCS*, 2013.

[95] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," *CoRR*, vol. abs/1706.04972, 2017. [Online]. Available: http://arxiv.org/abs/1706.04972

[96] Y. Xiao, K. Thulasiraman, G. Xue, and A. Jüttner, "The constrained shortest path problem: Algorithmic approaches and an algebraic study with generalization."

[97] S. Irnich and G. Desaulniers, "Shortest path problems with resource constraints," in *Column generation*. Springer, 2005, pp. 33–65.

[98] R. Hassin, "Approximation schemes for the restricted shortest path problem," *Mathematics of Operations research*, vol. 17, no. 1, pp. 36–42, 1992.

[99] A. Jüttner, "On resource constrained optimization problems," in *In: 4th JapaneseHungarian Symposium on Discrete Mathematics and Its Applications*, 2005.

[100] *Wild Rydes Image Processing Workflow*, https://github.com/aws-samples/aws-serverless-workshops/tree/master/ImageProcessing.

[101] D. E. King, "Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, pp. 1755–1758, 2009.

[102] A. Juels, "Targeted advertising ... and privacy too," in *CT-RSA 2001*. Berlin, Heidelberg: Springer-Verlag, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=646139.680791 pp. 408–424.

[103] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems."

[104] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting {SGX} enclaves from practical side-channel attacks," in *ATC 18)*, 2018, pp. 227–240.

[105] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2015. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdf

[106] *Google Asylo*, https://github.com/google/asylo, Last accessed December 2018.

[107] *TF Trusted*, https://github.com/dropoutlabs/tf-trusted.

[108] *Tensorflow Model Library*, https://github.com/tensorflow/models/tree/master/research/slim.

[109] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015. [Online]. Available: http://arxiv.org/abs/1505.04597

[110] A. Dosovitskiy and T. Brox, "Inverting convolutional networks with convolutional networks," *CoRR*, vol. abs/1506.02753, 2015. [Online]. Available: http://arxiv.org/abs/1506.02753

[111] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 568–576. [Online]. Available: http://papers.nips.cc/paper/5353-two-stream-convolutional-networks-for-action-recognition-in-videos.pdf

[112] L. Wang, Y. Xiong, Z. Wang, and Y. Qiao, "Towards good practices for very deep two-stream convnets," *CoRR*, vol. abs/1507.02159, 2015. [Online]. Available: http://arxiv.org/abs/1507.02159